追求代码质量

wizardforcel

Published
with GitBook



目錄

追求代码质量

> 来源：追求代码质量

在这个系列中，Andrew Glover 将重点阐述有关保证代码质量的一些有时看上去有点神秘的东西。

追求代码质量: 对 Ajax 应用程序进行单元测试

使用 *GWT* 更轻松地测试异步应用程序

您可能从编写 Ajax 应用程序中获得了极大乐趣，但是对它们执行单元测试却着实让人头痛。在本文中，Andrew Glover 着手解决 Ajax 的弱点（其中之一），即应对异步 Web 应用程序执行单元测试的固有挑战。 幸运的是，他发现在 Google Web Toolkit 的帮助下，解决这个特殊的代码质量问题要比预想的容易。

Ajax 在近期无疑是 Web 开发界最时髦的字眼之一 —— 与 Ajax 相关的工具、框架、书籍以及 Web 站点的剧增就是该技术流行的最好证明。 此外，Ajax 应用程序也相当灵巧，不是吗？不过，像任何一个开发过 Ajax 应用程序的人证实的一样， 对 Ajax 执行测试真的很不方便。事实上，Ajax 的出现已经从根本上使得许多测试框架和工具失效，因为它们并没有针对异步 Web 应用程序测试进行设计！

有趣的是，某个支持 Ajax 的框架的开发人员注意到了这个限制， 并为此做了一些非常新颖的设计：内置的可测试性。除此之外，由于该框架简化了使用 Java™ 代码（而不是 JavaScript）创建 Ajax 应用程序，它的起点甚高，并且充分利用了 Java 平台上无可置疑的标准测试框架：JUnit。

我所论及的框架当然是非常流行的 Google Web Toolkit，也就是 GWT。在本文中，我将向您展示 GWT 如何实际地利用 Java 兼容性， 使 Ajax 应用程序的每个部分都能像与之对应的同步应用程序一样进行测试。

改进代码质量

别错过 Andrew Glover 的 代码质量讨论论坛，里面有关于代码语法、测试框架以及如何编写专注于质量的代码的帮助。

JUnit 和 GWTTestCase

因为与 GWT 有关的 Ajax 应用程序采用 Java 代码编写， 所以非常适合开发人员使用 JUnit 进行测试。 事实上，GWT 开发小组还为此创建了一个帮助器类 `GWTTestCase` ，扩展自 JUnit 的 3.8.1 `TestCase` 。 该基类添加了一些功能，可测试 GWT 代码并处理某些基础实现从而启动并运行 GWT 组件。

Google Web Toolkit

Google Web Toolkit 在 Java Web 开发社区的发布声势浩大， 同时也获得了与之相称的巨大轰动。GWT 为利用 Java 代码进行设计、构建和部署支持 Ajax 的 Web 应用程序提供了一种新颖的方式。Java Web 开发人员不再需要学习 JavaScript 并花费数个小时解决特定于浏览器的问题，他们可以直接进行与 Ajax 有关的富含信息的动态 Web 应用程序设计。

需要提醒的是： `GWTTestCase` 并非用来测试与 UI 相关的代码 —— 它是为了便于测试那些由 UI 交互触发 的异步问题。对 `GWTTestCase` 用途的误解使许多刚接触 GWT 的开发人员备受挫折， 因为他们期望能够用它方便地模拟用户界面，但最终发现这是徒劳的。

Ajax 组件有两个基本组成： 体验和功能，这些都被设计成异步方式。 图 1 演示了一个模拟 Web 表单的简单 Ajax 组件。 由于该组件支持 Ajax，表单的提交是异步执行的（即：无需重新载入与传统表单提交关联的页面）。

图 **1.** 一个支持 **Ajax** 的简单 **Web** 表单

Dictionary Service

Word: [ ] Submit

输入一个有效单词，单击组件的 **Submit** 按钮，将向服务器发送消息请求该单词的定义。 该定义通过回调异步返回，相应地插入到 Web 页面，如图 2 所示：

图 **2.** 单击 **Submit** 按钮后显示响应

Dictionary Service

Word: pugnacious Submit

inclined to quarrel or fight readily; quarrelsome; belligerent; combative.

功能性和集成测试

图 2 所示的交互测试可用于多个不同场景， 但是其中两种场景最为常见。从功能性观点考虑，您或许希望编写一个测试：填入表单值，单击 **Submit** 按钮，然后验证表单是否显示定义。 另外一个选择是集成测试，使您能够验证客户端代码的异步功能。GWT 的 `GWTTestCase` 正是 被设计用来执行此类测试。

需要牢记的是：在 `GWTTestCase` 测试用例环境下不可以进行用户界面测试。 在设计和构建 GWT 应用程序时，您必须清楚不要依赖用户界面 测试代码。 这种思路需要把交互代码从业务逻辑中分离出来， 正如您已经了解的，这是最佳的入门实践！

举例而言，重新查看图 1 和图 2 所示的 Ajax 应用程序。 该应用程序由四个逻辑部分构成： `TextBox` 用于输入目标单词， `Button` 用于执行单击，还有两个 `Label` （一个用于 `TextBox` ，另一个显示定义）。 实际 GWT 模块的初始方法如清单 1 所示， 但是您该如何测

试这段代码呢？

清单 **1.** 一个有效的 **GWT** 应用程序，但是如何测试它？

```
public class DefaultModule implements EntryPoint {

 public void onModuleLoad() {
   Button button = new Button("Submit");
   TextBox box = new TextBox();
   Label output = new Label();
   Label label = new Label("Word: ");

   HorizontalPanel inputPanel = new HorizontalPanel();
   inputPanel.setStyleName("input-panel");
   inputPanel.setVerticalAlignment(HasVerticalAlignment.ALIGN_MIDDLE);
   inputPanel.add(label);
   inputPanel.add(box);

   button.addClickListener(new ClickListener() {
    public void onclick(Widget sender) {
      String word = box.getText();
      WordServiceAsync instance = WordService.Util.getInstance();
       try {
         instance.getDefinition(word, new AsyncCallback() {

           public void onFailure(Throwable error) {
             Window.alert("Error occurred:" + error.toString());
           }

           public void onSuccess(Object retValue) {
             output.setText(retValue.toString());
           }
          });
        }catch(Exception e) {
          e.printStackTrace();
        }
    }
   });

   inputPanel.add(button);
   inputPanel.setCellVerticalAlignment(button,
     HasVerticalAlignment.ALIGN_BOTTOM);

   RootPanel.get("slot1").add(inputPanel);
   RootPanel.get("slot2").add(output);
   }
}
```

清单 1 的代码在运行时发生了严重的错误：它无法按照 JUnit 和 GWT 的 `GWTTestCase` 进行测试。 事实上，如果我试着为这段代码编写测试，从技术方面来说它可以运行，但是无法按照逻辑工作。考虑一下：您如何对这段代码进行验证？惟一可用于测试的 `public` 方法返回的是 `void` ， 那么，您怎么能够验证其功能的正确性呢？

如果我想以白盒方式验证这段代码，就必须分离业务逻辑和特定于用户界面的代码，这就需要进行重构。这本质上意味着把清单 1 中的代码分离到一个便于测试的独立方法中。 但是这并非听上去那么简单。很明显组件挂钩是通过 `onModuleLoad()` 方法实现， 但是如果我想强制其行为，可能 必须操纵某些用户界面（UI）组件。

分解业务逻辑和 UI 代码

第一步是为每个 UI 组件创建访问器方法， 如清单 2 所示。按照该方式，我可以在需要时获取它们。

清单 **2.** 向 **UI** 组件添加访问器方法使其可用

```
public class WordModule implements EntryPoint {

  private Label label;
  private Button button;
  private TextBox textBox;
  private Label outputLabel;

  protected Button getButton() {
   if (this.button == null) {
    this.button = new Button("Submit");
   }
   return this.button;
  }

  protected Label getLabel() {
   if (this.label == null) {
    this.label = new Label("Word: ");
   }
   return this.label;
  }

  protected Label getOutputLabel() {
   if (this.outputLabel == null) {
    this.outputLabel = new Label();
   }
   return this.outputLabel;
  }

  protected TextBox getTextBox() {
   if (this.textBox == null) {
    this.textBox = new TextBox();
    this.textBox.setVisibleLength(20);
   }
   return this.textBox;
  }
}
```

现在我实现了对所有与 UI 相关的组件的编程式访问（假设所有需要进行访问的类都在同一个包内）。以后我可能需要使用其中一种访问进行验证。我现在希望限制 使用访问器，如我已经指出的，这是因为 GWT 并非设计用来进行交互测试。所以，我不是真的要试图测试某个按钮实例是否被单击， 而是要测试 GWT 模块是否会对给定的单词调用服务器端代码，并且服务器端会返回一个有效定义。方法为将 `onModuleLoad()` 方法的定义获取逻辑推入（不是故意用双关语！）一个可测试方法中，如清单 3 所示：

清单 **3.** 重构的 **onModuleLoad** 方法委托给更易于测试的方法

```
public void onModuleLoad() {
  HorizontalPanel inputPanel = new HorizontalPanel();
  inputPanel.setStyleName("disco-input-panel");
  inputPanel.setVerticalAlignment(HasVerticalAlignment.ALIGN_MIDDLE);

  Label lbl = this.getLabel();
  inputPanel.add(lbl);

  TextBox txBox = this.getTextBox();
  inputPanel.add(txBox);

  Button btn = this.getButton();

  btn.addClickListener(new ClickListener() {
   public void onClick(Widget sender) {
     submitWord();
   }
  });

  inputPanel.add(btn);
  inputPanel.setCellVerticalAlignment(btn,
      HasVerticalAlignment.ALIGN_BOTTOM);

  if(RootPanel.get("input-container") != null) {
   RootPanel.get("input-container").add(inputPanel);
  }

  Label output = this.getOutputLabel();
  if(RootPanel.get("output-container") != null) {
   RootPanel.get("output-container").add(output);
  }
}
```

如清单 3 所示，我已经把 `onModuleLoad()` 的定义获取逻辑委托给 `submitWord` 方法，如清单 4 定义：

清单 **4.** 我的 **Ajax** 应用程序的实质！

```
protected void submitWord() {
  String word = this.getTextBox().getText().trim();
  this.getDefinition(word);
}

protected void getDefinition(String word) {
 WordServiceAsync instance = WordService.Util.getInstance();
 try {
   instance.getDefinition(word, new AsyncCallback() {

   public void onFailure(Throwable error) {
    Window.alert("Error occurred:" + error.toString());
   }

   public void onSuccess(Object retValue) {
    getOutputLabel().setText(retValue.toString());
   }
  });
 }catch(Exception e) {
   e.printStackTrace();
 }
}
```

`submitWord()` 方法又委托给 `getDefinition()` 方法，我可以用 JUnit 测试它。`getDefinition()` 方法从逻辑上独立于特定于 UI 的代码（对于绝大部分而言），并且可以在没有单击按钮的情况下得到调用。另一方面， 与异步应用程序有关的状态问题和 Java 语言的语义规则也规定了我不能在测试中完全 避免与 UI 相关的交互。仔细查看清单 4 中的代码，您能够发现激活异步回调的 `getDefinition()` 方法操纵了某些 UI 组件 —— 一个错误警告窗口以及一个 `Label` 实例。

我还可以通过获得输出 `Label` 实例的句柄，断言其文本是否是给定单词的定义，从而验证应用程序的功能。 在用 `GWTTestCase` 测试时，最好不要 尝试手工 强制改变组件状态，而应该让 GWT 完成这些工作。举例而言，在清单 4 中，我想验证对某个给定单词返回了其 正确定义并放入一个输出 `Label` 中。无需操作 UI 组件来设置这个单词；我只要直接调用 `getDefinition` 方法，然后断言 `Label` 具有对应定义。

既然我已经编写好了计划进行测试的 GWT 应用程序，我需要实际编写测试，这意味着设置 GWT 的 `GWTTestCase` 。

设置 GWTTestCase

若想从 `GWTTestCase` 的测试魔力中获益，需要遵守一些规则。 幸运的是，规则很简单：

- 所有用于实现测试的类和待测 GWT 模块必须位于同一个包内。
- 运行测试时，您必须至少传递一个 VM 参数，指明在哪种 GWT 模式（托管或 Web）下运行测试。
- 您必须实现 `getModuleName()` 方法，它返回一个 `String` ，表示您的 XML 模块文件。

最后，因为与服务器端实体通信的 Ajax 应用程序在本质上是异步的，GWT 还提供了 `Timer` 类 ， 以便延迟 JUnit，使异步行为在进行相关断言之前全部完成。

实现 getModuleName 和 Timer 类

我已经指出，我的测试集中于 `getDefinition()` 方法（如 清单 4 所示）。 您可以从代码看到，测试逻辑非常简单：传入一个单词（比如 *pugnacious*），然后验证相应的 `Label` 文本是否得到正确定义。很简单，对吗？但是不要忘记， `getDefinition()` 方法在 `AsyncCallback` 对象中具有某种相关的异步性。

`GWTTestCase` 类是一个抽象 类，因为它的 `getModuleName()` 方法就是这么声明的；因此，当您 扩展该类时，您需要实现 `getModuleName()` （除非您是在为框架创建自己的基抽象类）。模块名实际上就是您的 GWT XML 文件所在的包结构的名称去掉文件扩展名。举个例子，在本例中， 我有一个名为 WordModule.gwt.xml 的 XML 文件，它位于 一个目录结构如：com/acme/gwt。相应的， 模块的逻辑名称为 `com.acme.gwt.WordModule` ， 这会让您想到 Java 平台的普通包模式。

我已经得到一个模块名，可以开始定义测试用例了，如清单 5 所示：

清单 **5.** 您必须实现 **getModuleName** 方法并提供一个有效的名字

```
import com.google.gwt.junit.client.GWTTestCase;
import com.google.gwt.user.client.Timer;

public class WordModuleTest extends GWTTestCase {

 public String getModuleName() {
   return "com.acme.gwt.WordModule";
 }
}
```

到目前为止一切良好，但是我还没有执行任何测试！由于我的 Ajax 应用程序使用 `AsyncCallback` 对象，在通过测试用例调用 `getDefinition()` 方法时， 我必须强迫 JUnit 延迟运行；否则测试将由于没有任何响应而失败。这就要用到 GWT 的 `Timer` 类。`Timer` 使我能够重写 `getDefinition()` 的 `run` 方法，在 `Timer` 内完成测试用例逻辑。（测试用例以独立线程运行，有效地阻塞 JUnit 完成整个测试用例）。

以我的测试为例，我将首先调用 `getDefinition()` 方法，然后提供一个 `Timer` 的 `run()` 方法的实现。`run()` 方法得到输出 `Label` 实例的文本并验证是否是正确定义。 定义了 `Timer` 实例后，我就需要确定其何时运行，同时强制 JUnit 挂起直至 `Timer` 实例完成。也许听起来有点复杂，不必担心，因为实践起来非常简易。实际上，清单 6 展示了整个过程：

清单 **6.** 使用 **GWT** 轻松测试

```
public void testDefinitionValue() throws Exception {
 WordModule module = new WordModule();
 module.getDefinition("pugnacious");
 Timer timer = new Timer() {
   public void run() {
    String value = module.getOutputLabel().getText();
    String control = "inclined to quarrel or fight readily;...";
    assertEquals("should be " + control, control, value);
    finishTest();
   }
  };
  timer.schedule(200);
  delayTestFinish(500);
}
```

正如您所见，`Timer` 的 `run()` 方法 是我真正验证 Ajax 应用程序功能及其应用远程过程调用的地方。 请注意 run 方法的最后一步是调用 `finishTest()` 方法， 它意味着一切如预期运行，JUnit 可以不受阻塞正常运行。 在实践中，您可能会发现需要根据异步行为完成所需的时间调整延迟时间。 但用 JUnit 测试 GWT 应用程序的要点在于：您能够在无需 部署完整功能的 Web 应用程序的情况下测试它。因此，您能够更早地 并且更 频繁地 测试您的 GWT 应用程序。

运行 GWT 测试

使用 GWT 进行功能测试

像本文演示的这类简单 Ajax 应用程序可以 从功能角度进行验证， 使用包括 Selenium 在内的框架，它会驱动浏览器模拟实际用户行为。不过，要想用 Selenium 运行功能测试，您必须部署完整功能的 Web 应用程序。

前面我曾提到，如果您想实际运行您的 GWT JUnit 测试，您必须执行大量琐碎的工作来配置运行环境。比如说，要想通过 Ant 的 `junit` 任务运行我的测试，我就必须确保某些文件 位于类路径中并向低层 JVM 提供一个参数。特别是，在调用 `junit` 任务时， 我还要确保托管源文件（以及测试）的目录（或多个目录）位于类路径中，还要告诉 GWT 以何种模式运行。我倾向于使用 *hosted* 模式，这意味着要使用 *www-test* 标志 ，如清单 7 所示：

清单 **7.** 用 **Ant** 运行 **GWT** 测试

```
<junit dir="./" failureproperty="test.failure" printSummary="yes"
     fork="true" haltonerror="true">

 <jvmarg value="-Dgwt.args=-out www-test" />

 <sysproperty key="basedir" value="." />
 <formatter type="xml" />
 <formatter usefile="false" type="plain" />
 <classpath>
  <path refid="gwt-classpath" />
  <pathelement path="build/classes" />
  <pathelement path="src" />
  <pathelement path="test" />
 </classpath>
 <batchtest todir="${testreportdir}">
  <fileset dir="test">
   <include name="**/**Test.java" />
  </fileset>
 </batchtest>
</junit>
```

运行 GWT 测试现在转变成调用问题了。还需注意的是 GWT 测试属于轻量级测试， 所以我可以频繁运行测试，甚至是连续运行，就像我在一个持续集成环境（Continuous Integration）中一样。

结束语

在本文所示的 GWT 测试用例中，您已经看到用于验证 Ajax 应用程序所需的基本步骤。 您可以继续测试我的示例 GWT 应用程序，比如测试一些边界用例，但是我认为重点在于：如果使用包含测试特性的框架编写 Ajax 应用程序，测试要比想象中容易。

要对 GWT 应用程序进行良好测试（对绝大多数应用程序也适用），关键在于设计应用程序时要把测试一并考虑。还要注意 `GWTTestCase` 不是被用来进行交互测试的。 您不能使用 `GWTTestCase` 直接模拟用户。不过您能够以一种间接的方式用它来验证用户交互，正如本文中演示的那样。

追求代码质量: 使用 TestNG-Abbot 实现自动化 GUI 测试

使用 *fixture* 对象轻松验证 *GUI* 组件

TestNG-Abbot 是一种测试框架，它为 GUI 组件的测试带来了新的活力。本月，Andrew Glover 将带领您亲历使用 TestNG-Abbot 测试 GUI 过程中难度最大的部分，即理解用户场景的实现过程。一旦理解了它，您会发现将 GUI 组件隔离并使用框架所含的极其方便的 fixture 对象对其进行验证是多么地简单。

使用 Swing、AWT 和类似的技术构建用户界面通常会给开发人员进行测试带来挑战，原因如下：

- 底层图形框架的复杂性
- GUI 中表现形式和业务逻辑之间的耦合
- 缺乏直观的自动测试框架

当然，前两个原因并不新鲜 —— 图形框架本来就很复杂，而且向 GUI 应用程序添加业务功能总是会给测试造成麻烦。另一方面，过去几年中有许多方便的框架涌现出来，确实使 GUI 测试更加便利。

本月，我将介绍一种新的框架，它极大地减轻了 GUI 测试的痛苦。

TestNG-Abbot 简介

TestNG-Abbot 源自于两个成功的开发人员测试框架的结合：Abbot 和 TestNG。Abbot 是一种 JUnit 扩展框架，主要目的是使 GUI 组件实现编程隔离，它还提供了一种验证 GUI 行为的简易方法。举例来说，可以使用它来获取对按钮组件的引用，使用编程的方法点击按钮，然后检验其操作。Abbot 还附带了一个脚本记录器，使用它就能够以 XML 格式布设测试场景，可以通过编程的方式运行它。

希望改善代码质量吗？

那么千万不要错过 Andrew 的 改善 Java 代码质量论坛，在那里可以学到关于代码度量标准、测试框架以及编写质量为先的代码的第一手知识。

在本系列中，我已经介绍了一些关于 TestNG 的内容，这里将继续介绍 TestNG。基本上，TestNG 是 JUnit 的一个替代物。除了所有预期的功能外，它还增加了一些额外功能。正如我在其他文章中提到的一样，TestNG 特别适合于更高层次的测试，其中，它可以用来测试依赖

关系并只返回失败了的测试 —— 简而言之，在测试 GUI 时，这类型测试非常方便。（参见 Resources 中有关 TestNG 的更多内容。）

它的起源就如此让人印象深刻，所以 TestNG-Abbot 成为测试工具中的神童就没什么好奇怪的了。同 Abbot 一样，TestNG-Abbot 使 GUI 组件能够进行编程隔离。同时，它使用了 TestNG 的断言，将 GUI 操作细节提取到了简单的 fixture 中，后者能够公开验证方法。如能正确使用，TestNG-Abbot 的直观的 fixture 类能够使 GUI 测试如同从小男孩手里偷一块糖一样简单。（当然，您不会想那么干的！）

直观的 fixture 类

TestNG-Abbot 的当前版本支持七种 fixture 类型，其中一种类型用于操作按钮、菜单标签以及文本项组件，如文本字段。此外，这些 fixture 类型根据名字在逻辑上链接到了测试中的代码（即 GUI 组件）。这使得 GUI 和其测试实现了松耦合，这样做至少有以下两个好处：

- 测试不会依赖于特定位置的 GUI 组件 —— 这样无需中断测试就可对其进行移动。
- 可以在早期进行测试，并且不会受到开发期间布局和外观改变的影响。

虽然目前只支持七种 fixture 类型，很快就会支持其他 fixture 类型。更多的 fixture 类型只会增加 TestNG-Abbot 在编程验证 GUI 方面的高效性。

GUI 验证不再普通！

虽然 TestNG-Abbot 使得验证 GUI 的过程更加简单，这并不意味着这个过程很简单。必须使 GUI 测试区别于单元或组件测试。验证 GUI 中业务规则的过程变成了对用户场景进行验证；或者，换种说法，GUI 测试包括验证可见状态的改变。

比方说，如果按下了定单输入 GUI 上的保存按钮，业务规则会保证命令的内容被保存到数据库中。然而，在一个用户场景中，会保证成功的状态信息被插入了按钮下 —— 这正是使用 TestNG-Abbot 能够编写的测试。事实上，如果 GUI 设计良好的话，可以测试被保存到数据库中的命令内容而无需 测试 GUI。接着您还可以同时并及早地编写这两个特别的测试。

加油！

记住 TestNG-Abbot 并不妨碍端对端测试（end-to-end）。可以轻松将 TestNG-Abbot 和 DbUnit 结合在一起，比如，创建一个同时验证用户场景和业务规则的可重复测试。

Word Finder GUI

为了使您了解 TestNG-Abbot 工作原理，我创建了一个简单的 GUI，它执行一种功能 —— 在底层字典（也就是一个数据库）中查阅一个给定的单词并显示其释义。不管该应用程序实际的代码如何，测试该 GUI 用户场景包括三个步骤：

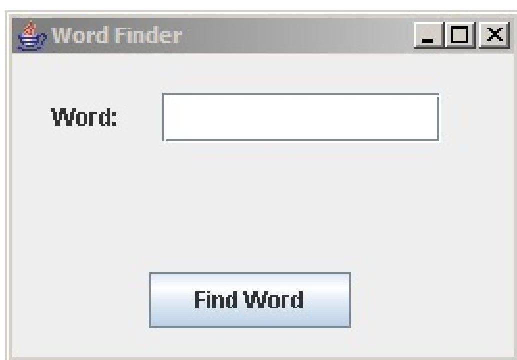
1. 在文本框中输入一个单词。
2. 单击 **Find Word** 按钮。
3. 验证是否给出了该单词释义。

当然，也存在一些极端的例子，比如一个用户按下了 Find Word 按钮但没有输入单词，或者，用户输入了一个无效的单词。我将通过一些其他的测试案例说明如何处理这类场景。

了解 GUI

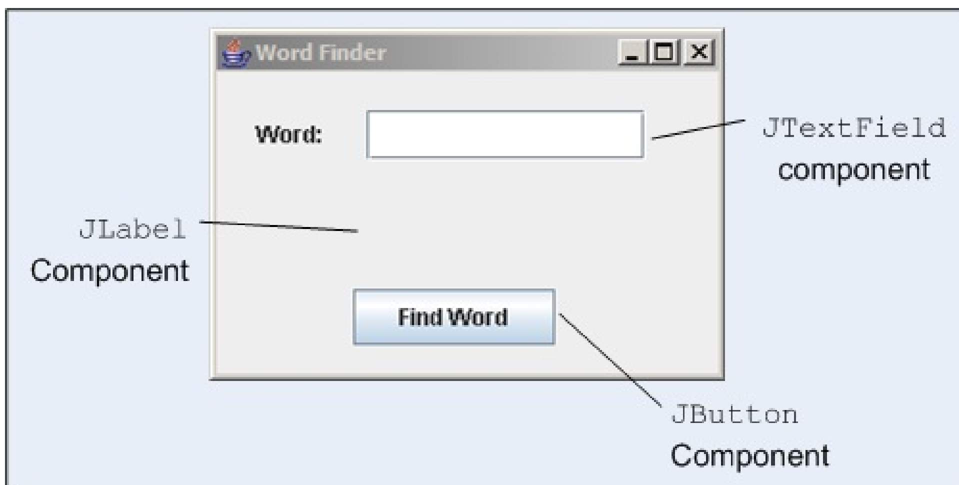
图 1 显示了已启动的 Word Finder GUI。记住该 GUI 之所以简单只有一个原因：它演示了 TestNG-Abbot 的三个 fixture 类以及一些要引导的测试用例！

图 **1. Word Finder GUI**



当使用 TestNG-Abbot 进行测试时，应该首先检查 GUI 的组件。Word Finder GUI 由图 2 所示的三个组件组成：

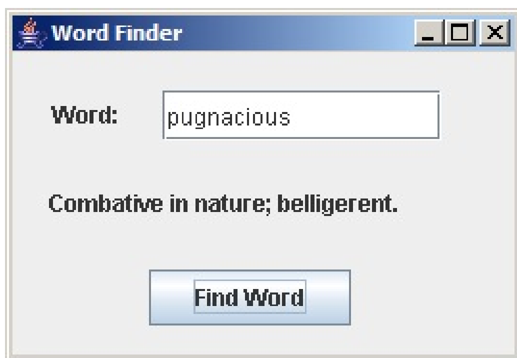
图 **2. Word Finder GUI** 的组件



如您所见，Word Finder GUI 由一个 `JTextField`（用来输入要查询的单词）、一个 `JButton`（使 GUI 从字典数据库中获取释义）和一个 `JEditorPane`（显示释义）组成。

在良好的场景中，如果我输入 *pugnacious* 然后单击 Find Word 按钮，`JEditorPane` 将显示 “Combative in nature; belligerent”，如图 3 所示：

图 **3.** 良好的场景 —— 工作良好！



使用 **TestNG-Abbot** 进行测试

要开始使用 TestNG-Abbot，需要创建一个常规的测试 fixture，它将使用 TestNG 的 `BeforeMethod` 和 `AfterMethod` 注释为您的 GUI 创建一个实例。TestNG-Abbot 框架附带了一个方便的 `AbbotFixture` 对象，它简化了 GUI 组件的使用，实际上也引导了整个测试过程。要在测试 fixture 中使用该对象，需要在测试前将一个 GUI 实例传递给 fixture 对象的 `showWindow()` 方法，然后使用名为 `cleanUp()` 的方法对 fixture 进行清理。

在清单 1 中，我创建了一个 TestNG 测试（实际上并没有对其做任何测试），该测试在 fixture 中使用 TestNG-Abbot 的 `AbbotFixture` 对象来存放 Word Finder GUI 的实例。

清单 **1.** 使用 **AbbotFixture** 对象定义 **WordFindGUITest**

```
public class WordFindGUITest {
 private AbbotFixture fixture;

 @BeforeMethod
 private void initializeGUI() {
  fixture = new AbbotFixture();
  fixture.showWindow(new WordFind(), new Dimension(269, 184));
 }

 @AfterMethod
 public void tearDownGUI() {
  fixture.cleanUp();
 }
}
```

由于 Word Finder GUI 的用户可见的行为会影响 图 2 所示的三个组件，需要通过编程对其进行调整，以确保工作能正常进行。比如，验证 图 3 演示的良好的场景，需要执行下面三个步骤：

1. 获得对 `JTextField` 的引用并向其添加一些文本。
2. 获得 `JButton` 的句柄并单击它。
3. 获得对 `JLabel` 组件的引用并检验是否显示了正确的释义。

检验良好的场景

使用 TestNG-Abbot，可以通过这三个方便的 fixture 类型执行上面所属的三个步骤：`TextComponentFixture` 用于 `JTextField`；`ButtonFixture` 用于 **Find Word** 按钮；`LabelFixture` 用来验证 `JLabel` 中特定的文本。

清单 2 显示了用于验证 图 3 中演示的内容是否可以正常工作的代码：

清单 **2.** 测试一个良好场景

```
@Test
public void assertDefinitionPresent() {
 TextComponentFixture text1 = new TextComponentFixture(this.fixture,
   "wordValue");
 text1.enterText("pugnacious");

 ButtonFixture bfix = new ButtonFixture(this.fixture, "findWord");
 bfix.click();

 LabelFixture fix = new LabelFixture(this.fixture, "definition");
 fix.shouldHaveThisText("Combative in nature; belligerent.");
}
```

注意 fixture 对象通过一个逻辑名称和特定的 GUI 组件连接在一起。例如，在 Word Finder GUI 中，通过编程将 `JButton` 对象与 “findWord” 名称联系起来。请注意在定义按钮时，我是如何通过调用组件的 `setName()` 方法做到这点的，如清单 3 所示：

清单 **3.** 定义 **Find Word** 按钮

```
findWordButton = new JButton();
findWordButton.setBounds(new Rectangle(71, 113, 105, 29));
findWordButton.setText("Find Word");
findWordButton.setName("findWord");
```

同样要注意，在 清单 2 中，我是如何通过将 “findWord” 名称传递给 TestNG-Abbot 的 `ButtonFixture` 对象而获得对按钮的引用。“单击” 按钮（调用 `click` 方法）然后使用 TestNG-Abbot 的 `LabelFixture` 对象插入单词的释义，多么酷！不过不要就此满足。

测试意外场景

当然，如果我非常希望验证我的 Word Finder GUI，我必须确保在用户执行意外操作时 —— 程序能够正常工作，比如在输入单词之前按下 Find Word 按钮，或者情况更糟，比如他们输入了一个无效的单词。举例来说，如果用户没有向文本字段输入内容，GUI 应该显示特定的信息，如清单 4 所示：

图 **4.** 糟糕的极端例子



当然，使用 TestNG-Abbot 测试这种情况非常简单，不是吗？我所做的仅仅是将空值传送到 `TextComponentFixture` 中，按下按钮（通过对 `ButtonFixture` 使用 `click` 方法）并插入 “Please enter a valid word” 响应！

清单 **4.** 测试一个极端例子：如果有人没有输入单词就按下了按钮该怎么办？

```
@Test
public void assertNoWordPresentInvalidText() {
 TextComponentFixture text1 = new TextComponentFixture(this.fixture,
   "wordValue");
 text1.enterText("");

 ButtonFixture bfix = new ButtonFixture(this.fixture, "findWord");
 bfix.click();

 LabelFixture fix = new LabelFixture(this.fixture, "definition");
 fix.shouldHaveThisText("Please enter a valid word");
}
```

如清单 4 所示，一旦理解了获得所需 GUI 组件的引用时，事情并不是很困难。最后一步是检验其他 糟糕的极端例子 —— 输入了无效的单词。这个过程与 清单 1 和 清单 3 非常相似：仅仅是将所需的 `String` 传递到 `TextComponentFixture` 对象，单击，然后插入特定的文本。如清单 5 所示：

清单 **5.** 轻松验证另一个极端例子！

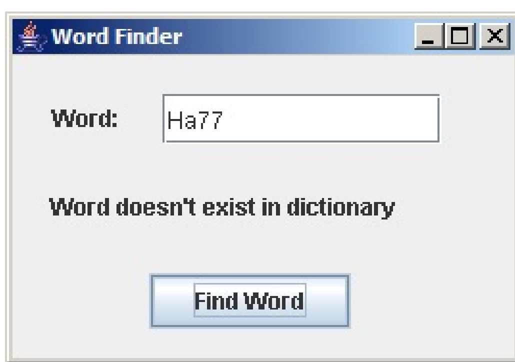
```
@Test
public void assertNoWordPresentInvalidText() {
 TextComponentFixture text1 = new TextComponentFixture(this.fixture,
   "wordValue");
 text1.enterText("Ha77");

 ButtonFixture bfix = new ButtonFixture(this.fixture, "findWord");
 bfix.click();

 LabelFixture fix = new LabelFixture(this.fixture, "definition");
 fix.shouldHaveThisText("Word doesn't exist in dictionary");
}
```

清单 5 很好地验证了图 5 演示的功能，难道您不这样认为吗？

图 **5.** 输入了无效单词



真不错！我们已经使用 TestNG-Abbot 轻而易举地验证了三种不同的用户场景。对于每种情况，我需要的只是被测试的组件的逻辑名称以及一系列步骤，以便创建场景。

继续测试 GUI

TestNG-Abbot 可能是测试工具中的新生儿，但它从其前辈那里继承了一些非常有用的特性。本文向您展示了如何使用 TestNG-Abbot 通过编程的方法将 GUI 组件隔离，然后使用 fixture 公开组件的验证方法。在这个过程中，您了解了对正常情况下的场景（所有事务都合乎逻辑）以及无法预见场景下（包括意外操作）进行测试是多么简单。总之，你只需要知道场景和组件在其中起到了作用。使用 TestNG-Abbot 方便的 fixture 对象可以很轻易地改变组件的行为。

追求代码质量: 用 AOP 进行防御性编程

OVal 省去了编写重复性条件的麻烦事

虽然防御性编程有效地保证了方法输入的条件，但如果在一系列方法中使用它，不免过于重复。本月，Andrew Glover 将向您展示通过一种更为容易的方式，即使用 AOP、契约式设计和一个便捷的叫做 OVal 的库，来向代码中添加可重用的验证约束条件。

开发人员测试的主要缺点是：绝大部分测试都是在理想的场景中进行的。在这些情况下并不会出现缺陷 —— 能导致出现问题的往往是那些边界情况。

什么是边界情况呢？比方说，把 `null` 值传入一个并未编写如何处理 `null` 值的方法中，这就是一种边界情况。大多数开发人员通常都不能成功测试这样的场景，因为这没多大意义。但不管有没有意义，发生了这样的情况，就会抛出一个 `NullPointerException` ，然后整个程序就会崩溃。

本月，我将为您推荐一种多层面的方法，来处理代码中那些不易预料的缺陷。尝试为应用程序整合进防御性编程、契约式设计和一种叫做 OVal 的易用的通用验证框架。

下载 OVal 和 AspectJ

要实现本文中描述的编程解决方案，需要下载 OVal 和 AspectJ。现在请从 参考资料 中下载这些技术，并照着那些例子做。

将敌人暴露出来

清单 1 中的代码为给定的 `Class` 对象（省去了 `java.lang.Object` ，因为所有对象都最终由它扩展）构建一个类层次。但如果仔细看的话，您会注意到一个有待发现的潜在缺陷，即该方法对对象值所做的假设。

清单 **1.** 不检验 **null** 的方法

```
public static Hierarchy buildHierarchy(Class clzz){

 Hierarchy hier = new Hierarchy();
 hier.setBaseClass(clzz);
 Class superclass = clzz.getSuperclass();

 if(superclass != null && superclass.getName().equals("java.lang.Object")){
  return hier;
 }else{
  while((clzz.getSuperclass() != null) &&
    (!clzz.getSuperclass().getName().equals("java.lang.Object"))){
     clzz = clzz.getSuperclass();
     hier.addClass(clzz);
  }
  return hier;
 }
}
```

刚编好这个方法，我还没注意到这个缺陷，但由于我狂热地崇拜开发人员测试，于是我编写了一个使用 TestNG 的常规测试。而且，我还利用了 TestNG 方便的 `DataProvider` 特性，借助该特性，我创建了一个通用的测试用例并通过另一个方法来改变它的参数。运行清单 2 中定义的测试用例会产生两个通过结果！一切都运转良好，不是吗？

清单 **2.** 验证两个值的 **TestNG** 测试

```
import java.util.Vector;
import static org.testng.Assert.assertEquals;
import org.testng.annotations.DataProvider;
import org.testng.annotations.Test;

public class BuildHierarchyTest {

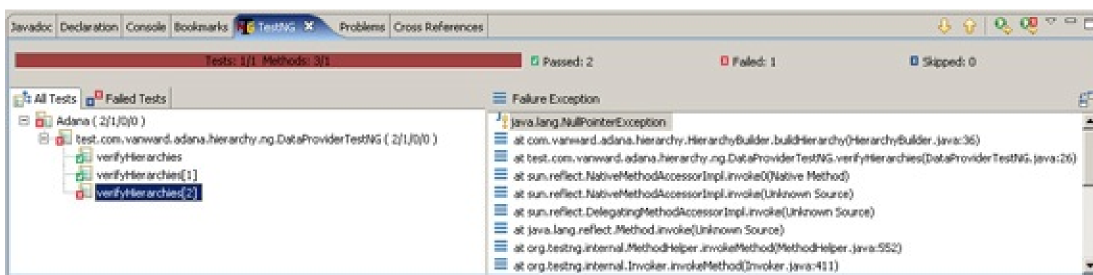
 @DataProvider(name = "class-hierarchies")
 public Object[][] dataValues(){
  return new Object[][]{
   {Vector.class, new String[] {"java.util.AbstractList",
      "java.util.AbstractCollection"}},
   {String.class, new String[] {}}
  };
 }

 @Test(dataProvider = "class-hierarchies"})
 public void verifyHierarchies(Class clzz, String[] names) throws Exception{
  Hierarchy hier = HierarchyBuilder.buildHierarchy(clzz);
  assertEquals(hier.getHierarchyClassNames(), names, "values were not equal");
 }
}
```

至此，我还是没有发现缺陷，但一些代码问题却困扰着我。如果有人不经意地为 `Class` 参数传入一个 `null` 值会怎么样呢？清单 1 中第 4 行的 `clzz.getSuperclass()` 调用会抛出一个 `NullPointerException` ，是这样吗？

测试我的理论很容易；甚至都不用从头开始。仅仅把 `{null, null}` 添加到初始 `BuildHierarchyTest` 的 `dataValues` 方法中的多维 `Object` 数组中，然后再次运行它。我定会得到如图 1 所示的 `NullPointerException` ：

图 **1.** 可怕的 **NullPointerException**



参见这里的 全图。

关于静态分析

诸如 FindBugs 等静态分析工具通过将字节码和一系列 bug 模式相匹配来检验类或 JAR 文件从而寻找潜在问题。针对样例的代码运行 FindBugs 不会 揭示出清单 1 中找到的 `NullPointerException` 。

防御性编程

一旦出现这个问题，下一步就是要拿出对抗的策略。问题是我控制不了这个方法能否接收这种输入。对于这类问题，开发人员通常会使用防御性编程技术，该技术专门用来在发生摧毁性后果前捕捉潜在错误。

对象验证是处理不确定性的一项经典的防御性编程策略。相应地，我会添加一项检验来验证 `clzz` 是否为 `null` ，如清单 3 所示。如果其值最终为 `null` ，我就会抛出一个 `RuntimeException` 来警告他人注意这个潜在问题。

清单 **3.** 添加验证 **null** 值的检验

```
public static Hierarchy buildHierarchy(Class clzz){

 if(clzz == null){
  throw new RuntimeException("Class parameter can not be null");
 }

 Hierarchy hier = new Hierarchy();
 hier.setBaseClass(clzz);

 Class superclass = clzz.getSuperclass();

 if(superclass != null && superclass.getName().equals("java.lang.Object")){
  return hier;
 }else{
  while((clzz.getSuperclass() != null) &&
    (!clzz.getSuperclass().getName().equals("java.lang.Object"))){
     clzz = clzz.getSuperclass();
     hier.addClass(clzz);
  }
  return hier;
 }
}
```

很自然，我也会编写一个快速测试用例来验证我的检验是否真能避免 `NullPointerException`，如清单 4 所示：

清单 **4.** 验证 **null** 检验

```
@Test(expectedExceptions={RuntimeException.class})
public void verifyHierarchyNull() throws Exception{
 Class clzz = null;
 HierarchyBuilder.buildHierarchy(null);
}
```

在本例中，防御性编程似乎解决了问题。但仅依靠这项策略会存在一些缺陷。

防御的缺陷

关于断言

清单 3 使用一个条件来验证 `clzz` 的值，实际上 `assert` 也同样好用。使用断言，无需指定条件，也不需要指定异常语句。在启用了断言的情况下，防御性编程的关注点全部由 JVM 处理。

尽管防御性编程有效地保证了方法的输入条件，但如果在一系列方法中使用它，不免过于重复。熟悉面向方面编程（或 AOP）的人们会把它认为是横切关注点，这意味着防御性编程技术横跨了代码库。许多不同的对象都采用这些语法，尽管从纯面向对象的观点来看这些语法跟对象毫不相关。

而且，横切关注点开始渗入到契约式设计（DBC）的概念中。DBC 是这样一项技术，它通过在组件的接口显式地陈述每个组件应有的功能和客户机的期望值来确保系统中所有的组件完成它们应尽的职责。从 DBC 的角度讲，组件应有的功能被认为是后置条件，本质上就是组件的责任，而客户机的期望值则普遍被认为是前置条件。另外，在纯 DBC 术语中，遵循 DBC 规则的类针对其将维护的内部一致性与外部世界有一个契约，即人所共知的类不变式。

契约式设计

我在以前的一篇关于用 Nice 编程的文章中介绍过 DBC 的概念，Nice 是一门与 JRE 兼容的面向对象编程语言，它的特点是侧重于模块性、可表达性和安全性。有趣的是，Nice 并入了功能性开发技术，其中包括了一些在面向方面编程中的技术。功能性开发使得为方法指定前置条件和后置条件成为可能。

尽管 Nice 支持 DBC，但它与 Java™ 语言完全不同，因而很难将其用于开发。幸运的是，很多针对 Java 语言的库也都为 DBC 提供了方便。每个库都有其优点和缺点，每个库在 DBC 内针对 Java 语言进行构建的方法也不同；但最近的一些新特性大都利用了 AOP 来更多地将 DBC 关注点包括进来，这些关注点基本上就相当于方法的包装器。

前置条件在包装过的方法执行前击发，后置条件在该方法完成后击发。使用 AOP 构建 DBC 结构的一个好处（请不要同该语言本身相混淆！）是：可以在不需要 DBC 关注点的环境中将这些结构关掉（就像断言能被关掉一样）。以横切的方式对待安全性关注点的真正妙处是：可以有效地重用 这些关注点。众所周知，重用是面向对象编程的一个基本原则。AOP 如此完美地补充了 OOP 难道不是一件极好的事情吗？

结合了 OVal 的 AOP

OVal 是一个通用的验证框架，它通过 AOP 支持简单的 DBC 结构并明确地允许：

- 为类字段和方法返回值指定约束条件
- 为结构参数指定约束条件
- 为方法参数指定约束条件

此外，OVal 还带来大量预定义的约束条件，这让创建新条件变得相当容易。

由于 OVal 使用 AspectJ 的 AOP 实现来为 DBC 概念定义建议，所以必须将 AspectJ 并入一个使用 OVal 的项目中。对于不熟悉 AOP 和 AspectJ 的人们来说，好消息是这不难实现，且使用 OVal （甚至是创建新的约束条件）并不需要真正对方面进行编码，只需编写一个简单的自引导程序即可，该程序会使 OVal 所附带的默认方面植入您的代码中。

在创建这个自引导程序方面前，要先下载 AspectJ。具体地说，您需要将 `aspectjtools` 和 `aspectjrt` JAR 文件并入您的构建中来编译所需的自引导程序方面并将其编入您的代码中。

自引导 AOP

下载了 AspectJ 后，下一步是创建一个可扩展 OVal `GuardAspect` 的方面。它本身不需要做什么，如清单 5 所示。请确保文件的扩展名以 .aj 结束，但不要试着用常规的 `javac` 对其进行编译。

清单 **5. DefaultGuardAspect** 自引导程序方面

```
import net.sf.oval.aspectj.GuardAspect;

public aspect DefaultGuardAspect extends GuardAspect{
 public DefaultGuardAspect(){
  super();
 }
}
```

AspectJ 引入了一个 Ant 任务，称为 `iajc`，充当着 `javac` 的角色；此过程对方面进行编译并将其编入主体代码中。在本例中，只要是我指定了 OVal 约束条件的地方，在 OVal 代码中定义的逻辑就会编入我的代码，进而充当起前置条件和后置条件。

请记住 `iajc` 代替了 `javac`。例如，清单 6 是我的 Ant build.xml 文件的一个代码片段，其中对代码进行了编译并把通过代码标注发现的所有 OVal 方面编入进来，如下所示：

清单 **6.** 用 **AOP** 编译的 **Ant** 构建文件片段

```
<target name="aspectjc" depends="get-deps">

 <taskdef resource="org/aspectj/tools/ant/taskdefs/aspectjTaskdefs.properties">
  <classpath>
   <path refid="build.classpath" />
  </classpath>
 </taskdef>

 <iajc destdir="${classesdir}" debug="on" source="1.5">
  <classpath>
   <path refid="build.classpath" />
  </classpath>
  <sourceroots>
   <pathelement location="src/java" />
   <pathelement location="test/java" />
  </sourceroots>
 </iajc>

</target>
```

为 OVal 铺好了路、为 AOP 过程做了引导之后，就可以开始使用 Java 5 标注来为代码指定简单的约束条件了。

OVal 的可重用约束条件

用 OVal 为方法指定前置条件必须对方法参数进行标注。相应地，当调用一个用 OVal 约束条件标注过的方法时，OVal 会在该方法眞正执行前 验证该约束条件。

在我的例子中，我想要指定当 `Class` 参数的值为 `null` 时，`buildHierarchy` 方法不能被调用。OVal 通过 `@NotNull` 标注支持此约束条件，该标注在方法所需的所有参数前指定。也要注意，任何想要使用 OVal 约束条件的类也必须在类层次上指定 `@Guarded` 标注，就像我在清单 7 中所做的那样：

清单 **7. OVal** 约束条件

```
import net.sf.oval.annotations.Guarded;
import net.sf.oval.constraints.NotNull;

@Guarded
public class HierarchyBuilder {

 public static Hierarchy buildHierarchy(@NotNull Class clzz){

  Hierarchy hier = new Hierarchy();
  hier.setBaseClass(clzz);

  Class superclass = clzz.getSuperclass();

  if(superclass != null && superclass.getName().equals("java.lang.Object")){
   return hier;
  }else{
   while((clzz.getSuperclass() != null) &&
     (!clzz.getSuperclass().getName().equals("java.lang.Object"))){
       clzz = clzz.getSuperclass();
       hier.addClass(clzz);
    }
   return hier;
  }
 }
}
```

通过标注指定这个约束条件意味着我的代码不再会被重复的条件弄得乱七八糟，这些条件检查 `null` 值，并且一旦找到该值就会抛出异常。现在这项逻辑由 OVal 处理，且处理的方法有些相似 —— 事实上，如果违反了约束条件，OVal 会抛出一个 `ConstraintsViolatedException`，它是 `RuntimeException` 的子类。

当然，我下一步就要编译 `HierarchyBuilder` 类和 清单 5 中相应的 `DefaultGuardAspect` 类。我用 清单 6 中的 `iajc` 任务来实现这一目的，这样我就能把 OVal 的行为编入我的代码中了。

接下来，我更新 清单 4 中的测试用例来验证是否抛出了一个 `ConstraintsViolatedException`，如清单 8 所示：

清单 **8.** 验证是否抛出了 **ConstraintsViolatedException**

```
@Test(expectedExceptions={ConstraintsViolatedException.class})
public void verifyHierarchyNull() throws Exception{
 Class clzz = null;
 HierarchyBuilder.buildHierarchy(clzz);
}
```

指定后置条件

正如您所见，指定前置条件其实相当容易，指定后置条件的过程也是一样。例如，如果我想对所有调用 `buildHierarchy` 的程序保证它不会返回 `null` 值（这样，这些调用程序就不需要再检查这个了），我可以在方法声明之上放置一个 `@NotNull` 标注，如清单 9 所示：

清单 **9. OVal** 中的后置条件

```
@NotNull
public static Hierarchy buildHierarchy(@NotNull Class clzz){
 //method body
}
```

当然，`@NotNull` 绝不是 OVal 提供的惟一约束条件，但我发现它能非常有效地限制这些令人讨厌的 `NullPointerException`，或至少能够快速地暴露 它们。

更多的 OVal 约束条件

OVal 也支持在方法调用前或后对类成员进行预先验证。这种机制具有限制针对特定约束条件的重复条件测试的好处，如集合大小或之前讨论过的非 `null` 的情况。

例如，在清单 10 中，我使用 `HierarchyBuilder` 定义了一个为类层次构建报告的 Ant 任务。请注意 `execute()` 方法是如何调用 `validate` 的，后者会依次验证 `fileSet` 类成员是否含值；如果不含，会抛出一个异常，因为没有了要评估的类，该报告不能运行。

清单 **10.** 带条件检验的 **HierarchyBuilderTask**

```
public class HierarchyBuilderTask extends Task {
 private Report report;
 private List fileSet;

 private void validate() throws BuildException{
  if(!(this.fileSet.size() > 0)){
   throw new BuildException("must supply classes to evaluate");
  }
  if(this.report == null){
   this.log("no report defined, printing XML to System.out");
  }
 }

 public void execute() throws BuildException {
  validate();
  String[] classes = this.getQualifiedClassNames(this.fileSet);
  Hierarchy[] hclz = new Hierarchy[classes.length];

  try{
   for(int x = 0; x < classes.length; x++){
    hclz[x] = HierarchyBuilder.buildHierarchy(classes[x]);
   }
   BatchHierarchyXMLReport xmler = new BatchHierarchyXMLReport(new Date(), hclz);
   this.handleReportCreation(xmler);
  }catch(ClassNotFoundException e){
   throw new BuildException("Unable to load class check classpath! " + e.getMessage());
  }
 }
//more methods below....
}
```

因为我用的是 OVal，所以我可以完成下列任务：

- 对 `fileSet` 类成员指定一个约束条件，确保使用 `@Size` 标注时其大小总是至少为 1 或

更大。

- 确保在使用 `@PreValidateThis` 标注调用 `execute()` 方法前 验证这个约束条件。

这两步让我能够有效地去除 `validate()` 方法中的条件检验，让 OVal 为我完成这些，如清单 11 所示：

清单 **11.** 经过改进、无条件检验的 **HierarchyBuilderTask**

```
@Guarded
public class HierarchyBuilderTask extends Task {
 private Report report;

 @Size(min = 1)
 private List fileSet;

 private void validate() throws BuildException {
  if (this.report == null) {
   this.log("no report defined, printing XML to System.out");
  }
 }

 @PreValidateThis
 public void execute() throws BuildException {
  validate();
  String[] classes = this.getQualifiedClassNames(this.fileSet);
  Hierarchy[] hclz = new Hierarchy[classes.length];

  try{
   for(int x = 0; x < classes.length; x++){
    hclz[x] = HierarchyBuilder.buildHierarchy(classes[x]);
   }
   BatchHierarchyXMLReport xmler = new BatchHierarchyXMLReport(new Date(), hclz);
   this.handleReportCreation(xmler);
  }catch(ClassNotFoundException e){
   throw new BuildException("Unable to load class check classpath! " + e.getMessage());
  }
 }
 //more methods below....
}
```

清单 11 中的 `execute()` 一经调用（由 Ant 完成），OVal 就会验证 `fileSet` 成员。如果其为空，就意味着没有指定任何要评估的类，就会抛出一个 `ConstraintsViolatedException`。这个异常会暂停这一过程，就像初始代码一样，只不过初始代码会抛出一个 `BuildException`。

结束语

防御性编程结构阻止了一个又一个缺陷，但这些结构本身却不免为代码添加了重复的逻辑。把防御性编程技术和面向方面编程（通过契约式设计）联系起来是抵御所有重复性代码的一道坚强防线。

OVal 并不是惟一可用的 DBC 库，事实上其 DBC 结构对比其他框架来说是相当有限的（例如，它未提供指定类不变式的简易方法）。从另一方面讲，OVal 很容易使用，对约束条件也有很大的选择余地，若想要花少量力气就可向代码添加验证约束条件，它无疑是个上佳之选。另外，用 OVal 创建定制约束条件也相当简单，所以请不要再添加条件检验了，尽情享用 AOP 吧！

追求代码质量: 探究 XMLUnit

一种用于测试 *XML* 文档的 *JUnit* 扩展框架

Java™ 开发人员一般都很善于解决问题，所以由 Java 开发人员提出更容易的方法用以验证 XML 文档是很自然的事。本月，Andrew 将向您介绍 XMLUnit，一个能满足您所有的 XML 验证需求的 JUnit 扩展框架。

在软件开发周期中，需要不时地验证 XML 文档的结构或内容。不管构建的是何种应用程序，测试 XML 文档都具有一定的挑战性，尤其是在没有相关工具的情况下就更是如此。

本月，我将首先向您说明为何不能 使用 `String` 比较来验证 XML 文档的结构和内容。之后，我会介绍 XMLUnit，一个由 Java 开发人员创建并可服务于 Java 开发人员的 XML 验证工具，向您展示如何使用它来验证 XML 文档。

古典的 String 比较

首先，假设您已经构建了一个应用程序，该应用程序可以输出代表对象依赖性报告的 XML 文档。对于给定的类和对应的过滤器的集合，会生成一个报告来输出类和类的依赖项（想象一下导入）。

清单 1 显示了用于给定类列表（ `com.acme.web.Widget` 和 `com.acme.web.Account` ）的报告，过滤器被设为忽略外部类，比如 `java.lang.String` ：

清单 **1.** 一个示例依赖性 **XML** 报告

```
<DependencyReport date="Sun Dec 03 22:30:21 EST 2006">
  <FiltersApplied>
    <Filter pattern="java|org"/>
    <Filter pattern="net."/>
  </FiltersApplied>
  <Class name="com.acme.web.Widget">
    <Dependency name="com.acme.resource.Configuration"/>
    <Dependency name="com.acme.xml.Document"/>
  </Class>
  <Class name="com.acme.web.Account">
    <Dependency name="com.acme.resource.Configuration"/>
    <Dependency name="com.acme.xml.Document"/>
  </Class>
</DependencyReport>
```

清单 1 很明显是由应用程序生成的；因而，第一层测试就是验证应用程序是否真能生成一个文档。一旦验证了这一点，就可以继续测试指定文档的其他三个方面：

- 结构
- 内容

- 指定内容

可以通过单独使用 JUnit 利用 `String` 比较处理上述前两个方面，如清单 2 所示：

清单**2.** 硬性验证 **XML**

```
public class XMLReportTest extends TestCase {

 private Filter[] getFilters(){
  Filter[] fltrs = new Filter[2];
  fltrs[0] = new RegexPackageFilter("java|org");
  fltrs[1] = new SimplePackageFilter("net.");
  return fltrs;
 }

 private Dependency[] getDependencies(){
  Dependency[] deps = new Dependency[2];
  deps[0] = new Dependency("com.acme.resource.Configuration");
  deps[1] = new Dependency("com.acme.xml.Document");
  return deps;
 }

 public void testToXML() {
  Date now = new Date();
  BatchDependencyXMLReport report =
   new BatchDependencyXMLReport(now, this.getFilters());

  report.addTargetAndDependencies(
    "com.acme.web.Widget", this.getDependencies());
  report.addTargetAndDependencies(
    "com.acme.web.Account", this.getDependencies());

  String valid = "<DependencyReport date=\"" + now.toString() + "\">"+
    "<FiltersApplied><Filter pattern=\"java|org\" /><Filter pattern=\"net.\" />"+
    "</FiltersApplied><Class name=\"com.acme.web.Widget\">" +
    " <Dependency name=\"com.acme.resource.Configuration\" />"+
    "<Dependency name=\"com.acme.xml.Document\" /></Class>"+
    "<Class name=\"com.acme.web.Account\">"+
    "<Dependency name=\"com.acme.resource.Configuration\" />"+
    "<Dependency name=\"com.acme.xml.Document\" />"+
    "</Class></DependencyReport>";

   assertEquals("report didn't match xml", valid, report.toXML());
 }
}
```

清单 2 中的测试有其他一些重大的缺陷 —— 而不仅仅是硬编码 `String` 比较那么简单。首先，测试并不真正可读。第二，它惊人的脆弱；一旦 XML 文档的格式改变（包括添加空格），与其尝试修复 `String` 本身，还不如粘贴进一个新的文档副本。最后，测试的本性会迫使您必须应付 `Date` 方面，虽然您并不想如此。

若想确保文档中第二个 `Class` 元素的 `name` 值是 `com.acme.web.Account` 又该如何呢？当然，您可以使用常规表达式或 `String` 搜索，但所需的工作量太大。这样看来，通过一个解析框架来操纵此 DOM 不是更有意义么？

XMLUnit 能否用于 TestNG？

XMLUnit 是一个 JUnit 扩展，但这并不意味着不能在 TestNG 使用它。只要它具有 API 而且此 API 支持委托同时不基于修饰器，那么您可以将几乎任何框架整合进 TestNG。

用 XMLUnit 进行测试

当您感觉自己为完成一项任务而努力过了头，您就可以想想解决此问题是否还有更容易的捷径可寻。如果所要解决的问题涉及的是编程式地验证 XML 文档，那么所应想到的解决方案就是 XMLUnit。

XMLUnit 是一种 JUnit 扩展框架，有助于开发人员测试 XML 文档。实际上，XMLUnit 是一种真正的 XML 测试的“多面手”：可以使用它来验证 XML 文档的结构、内容甚至该文档的指定部分。

最简单的做法是使用 XMLUnit 在逻辑上对比运行时 XML 文档和预定义的有效控制文件。本质上讲，这就是一种差异测试：假定一个 XML 文档是正确的，那么此应用程序在运行时是否会生成同样的东西？它是相对简单的一种测试，但也可以使用它来验证 XML 文档的结构和内容。也可以通过 XPath 的一点帮助来验证特定内容。

委托而非继承

首要原则是尽量避免测试用例继承。许多 JUnit 扩展框架，包括 XMLUnit，都提供可以通过继承得到的专门的测试用例来协助测试某一特定的架构。从框架继承来的测试用例都缺乏灵活性，这是 Java 平台的单一继承的范型所致。更多的时候，这些相同的 JUnit 扩展框架提供一个委托 API，此 API 可以更易于组合不同的框架，而无需采用严格的继承结构。

验证内容

可以通过委托或继承的方式使用 XMLUnit。作为最佳策略，我建议避免测试用例继承。另一方面，从 XMLUnit 的 `XMLTestCase` 继承确实可以提供一些方便的声明方法（这些方法不是 `静态` 的，因而也就不能像 JUnit的 `TestCase` 声明一样被静态引用）。

不管您如何选择使用 XMLUnit，都必须实例化 XMLUnit 的解析器。您可以通过 `System.setProperty` 调用实例化它们，也可以通过 `XMLUnit` 核心类上的一些方便的 `static` 方法对它们进行实例化。

一旦用所需要的不同的解析器实例化 XMLUnit 之后，就可以使用 `Diff` 类，这是从逻辑上对比两个 XML 文档所需的中心机制。在清单 3 中，我利用 XMLUnit 对 >testToXML test 做了一些改进：

清单 3. 改进后的 testToXML 测试

```
public class XMLReportTest extends TestCase {

  protected void setUp() throws Exception {
   XMLUnit.setControlParser(
     "org.apache.xerces.jaxp.DocumentBuilderFactoryImpl");
   XMLUnit.setTestParser(
     "org.apache.xerces.jaxp.DocumentBuilderFactoryImpl");
   XMLUnit.setSAXParserFactory(
     "org.apache.xerces.jaxp.SAXParserFactoryImpl");
   XMLUnit.setIgnoreWhitespace(true);
  }

  private Filter[] getFilters(){
   Filter[] fltrs = new Filter[2];
   fltrs[0] = new RegexPackageFilter("java|org");
   fltrs[1] = new SimplePackageFilter("net.");
   return fltrs;
  }

  private Dependency[] getDependencies(){
   Dependency[] deps = new Dependency[2];
   deps[0] = new Dependency("com.acme.resource.Configuration");
   deps[1] = new Dependency("com.acme.xml.Document");
   return deps;
  }

  public void testToXML() {
   BatchDependencyXMLReport report =
     new BatchDependencyXMLReport(new Date(1165203021718L),
       this.getFilters());

   report.addTargetAndDependencies(
     "com.acme.web.Widget", this.getDependencies());
   report.addTargetAndDependencies(
     "com.acme.web.Account", this.getDependencies());

   Diff diff = new Diff(new FileReader(
     new File("./test/conf/report-control.xml")),
     new StringReader(report.toXML()));

   assertTrue("XML was not identical", diff.identical());
  }
}
```

注意一下我是如何实例化 XMLUnit 的 `setControlParser` 、`setTestParser` 和 `setSAXParserFactory` 方法的。您可以为这些值使用任何兼容 JAXP 的解析器。还要注意我是用 `true` 调用 `setIgnoreWhitespace` 的 —— 这是一根救命稻草，相信我！否则，不一致的空白会导致很多故障。

用 Diff 比较

`Diff` 类支持两种比较：`identical` 和 `similar` 。如果所比较的文档在结构和值（如果设置了标志就忽略空白）方面都完全相同，那么它们就被认为是 *identical*；如果两个文档是完全相同的，那么它们也就很自然的是 *similar* 的。反之，却不一定。

例如，清单 4 是与清单 5 相似的一个简单的 XML 代码片段，但二者并不相同：

清单 **4.** 一个帐号 **XML** 片段

```
<account>
 <id>3A-00</id>
 <name>acme</name>
</account>
```

清单 5 中的 XML 片段与清单 4 中所示的 XML 片段有相同的逻辑文档。但 XMLUnit 并不认为二者是相同的，原因是二者的 `name` 和 `id` 元素是颠倒的。

清单 **5.** 一个相似的 **XML** 片段

```
<account>
 <name>acme</name>
 <id>3A-00</id>
</account>
```

相应地，我可以编写测试用例来验证 XMLUnit 的行为，如清单 6 所示：

清单 **6.** 用来验证相同性和相似性的测试

```
public void testIdenticalAndSimilar() throws Exception {
 String controlXML = "<account><id>3A-00</id><name>acme</name></account>";
 String testXML = "<account><name>acme</name><id>3A-00</id></account>";
 Diff diff = new Diff(controlXML, testXML);
 assertTrue(diff.similar());
 assertFalse(diff.identical());
}
```

相似和相同的 XML 文档之间的差异是很微小的；但若能验证两者却非常有用，例如在需要测试由不同应用程序或客户程序生成的文档的情况下。

验证结构

除了验证内容之外，您还需要验证 XML 文档的结构。在这种情况下，元素和属性的值并不重要 —— 您所关心的是结构。

还好，我还可以再次使用清单 3 中定义的测试用例来验证文档的结构，并可以有效忽略元素文本值和属性值。为实现此目的，我调用 `Diff` 类上的 `overrideDifferenceListener()` 并为它添加由 XMLUnit 提供的 `IgnoreTextAndAttributeValuesDifferenceListener` 。修改后的测试如清单 7 所示：

清单 **7.** 无需属性值验证 **XML** 结构

```
public void testToXMLFormatOnly() throws Exception{
 BatchDependencyXMLReport report =
   new BatchDependencyXMLReport(new Date(), this.getFilters());

 report.addTargetAndDependencies(
   "com.acme.web.Widget", this.getDependencies());
 report.addTargetAndDependencies(
   "com.acme.web.Account", this.getDependencies());

 Diff diff = new Diff(new FileReader(
   new File("./test/conf/report-control.xml")),
   new StringReader(report.toXML()));

 diff.overrideDifferenceListener(
   new IgnoreTextAndAttributeValuesDifferenceListener());
 assertTrue("XML was not similar", diff.similar());
}
```

相似但不相同！

当使用 `IgnoreTextAndAttributeValuesDifferenceListener` 类时，必须声明这两个文档是 `similar` 而非 `identical`。如果错误地调用了 `identical`，那么就需要处理属性值。

当然，DTD 的模式和 XML 模式都有助于 XML 结构验证，然而，有时文档并不需要引用它们 —— 在这些场景下，结构验证可能会很有用。同样，如果需要忽略特定的一些值（例如那些 `Date` 值），就可以实现 `DifferenceListener` 接口（正如 `IgnoreTextAndAttributeValuesDifferenceListener` 所做的一样）并提供一个定制实现。

XMLUnit 和 XPath

为实现 XML 测试的所有三个方面，XMLUnit 还可以借助 XPath 进行 XML 文档特定部分的验证。

例如，使用清单 1 所示相同的格式，我想验证由应用程序生成的第一个 `Class` 元素的 `name` 属性值是否是 `com.acme.web.Widget`。要实现此目的，我必须创建一个 XPath 表达式来导航到准确的位置；而且，XMLUnit 的 `XMLTestCase` 提供了一个方便的 `assertXpathExists()` 方法，这意味着我必须现在扩展 `XMLTestCase`。

清单 **8.** 使用 **XPath** 来验证准确的 **XML** 值

```
public void testToXMLFormatOnly() throws Exception{
 BatchDependencyXMLReport report =
   new BatchDependencyXMLReport(new Date(), this.getFilters());

 report.addTargetAndDependencies(
   "com.acme.web.Widget", this.getDependencies());
 report.addTargetAndDependencies(
   "com.acme.web.Account", this.getDependencies());

 assertXpathExists("//Class[1][@name='com.acme.web.Widget']",
  report.toXML());
}
```

如清单 8 所示，XMLUnit 和 XPath 一起协作提供了可以准确验证 XML 文档 的一种便捷机制，而不是进行大规模的差异测试。请记住要在 XMLUnit 内充分利用 XPath，您的测试用例必须要扩展 `XMLTestCase` 。如果熟悉 XPath 也会大有帮助！

XPath 是什么？

XPath 或 XML Path Language 是一种表达式语言，用来基于树表示定位 XML 文档的各部分。XPath 允许您导航 XML 文档并可以帮您选择文档值。

为何要舍近求远呢？

XMLUnit 是一种基于 Java 的开放源码工具，它使测试 XML 文档更为简单和灵活，而这是使用 `String` 比较所达不到的。使用 XMLUnit 进行差异测试所存在的惟一缺点是测试会依赖于文件系统来加载控制文档。在编写测试时，请务必考虑这一附加的依赖性。

虽然 XMLUnit 已经有段时间没有发布任何更新了，但它当前的特性集已经足够健壮来应对各种测试冲击，并且它用在这种情况下基本上是免费的！

追求代码质量: 用 JUnitPerf 进行性能测试

监控可伸缩性和性能的两个简单测试

在应用程序的开发周期中，性能测试常被放到最后考虑，这并不是因为它不重要，而是因为存在这么多未知变量，很难有效地测试。在本月的 追求代码质量 系列中，Andrew Glover 使性能测试成为开发周期的一部分，并介绍了两种简单的实现方法。

在应用程序的开发中，验证应用程序的性能几乎总处于次要的地位。请注意，我强调的是验证 应用程序的性能。应用程序的性能总是 首要考虑的因素，但开发周期中却很少包含对性能的验证。

由于种种原因，性能测试常被延迟到开发周期的后期。以我的经验，企业之所以在开发过程中不包含性能测试是因为，他们不知道对于正在进行开发的应用程序要期待什么。提出了一些（性能）指数，但这些指数是基于预期负载提出的。

发生下列两种情况之一时，性能测试就成为头等大事：

- 生产中出现显而易见的性能问题。
- 在同意付费之前 ，客户或潜在客户询问有关性能指数的问题。

本月，我将介绍两种简单的性能测试技术，在上述两种情况中的任何一种发生前进行测试。

改进代码质量

别错过 Andrew 的 讨论论坛，里面有关于代码语法、测试框架以及如何编写专注于质量的代码的帮助。

用 JUnitPerf 进行测试

在软件开发的早期阶段，使用 JUnit 很容易确定基本的低端性能指数。JUnitPerf 框架能够将测试快速地转化为简单的负载测试，甚至压力测试。

可使用 JUnitPerf 创建两种测试类型： `TimedTest` 和 `LoadTest` 。这两种类型都基于 Decorator 设计模式并利用 JUnit 的 `suite` 机制。 `TimedTest` 为测试样例创建一个（时间）上限 —— 如果超过这个时间，那么测试失败。 `LoadTest` 和计时器一起运行，它通过运行所需的次数（时间间隔由配置的计时器控制） ，在一个特定的测试用例上创建一个人工负载。

恰当的时限测试

JUnitPerf `TimedTest` 让您可以编写有相关时间限制的测试 —— 如果超过了该限度，就认为测试是失败的（即便测试逻辑本身实际上是成功的）。在测试对于业务致关重要的方法时，时限测试相比其他测试来说，在确定和监控性能指数方面很有帮助。甚至可以测试得更加细致一些，可以测试一系列方法来确保它们满足特定的时间限制。

例如，假设存在一个 Widget 应用程序，其中，特定的对于业务致关重要的方法（如 `createWidget()`）是严格的性能限制的测试目标。假设需要对执行该 `create()` 方法的功能方面进行性能测试。这通常会由不同的团队使用不同的工具在开发周期的后期加以确定，这通常不能指出精确的方法。但假设决定选择早期经常测试 方法取而代之。

创建 `TimedTest` 首先要创建一个标准的 JUnit 测试。换言之，将对 `TestCase` 或其派生类进行扩展，并编写一个以 `test` 开头的方法，如清单 1 所示：

清单 **1.** 简单的 **widget** 测试

```
public class WidgetDAOImplTest extends TestCase {
 private WidgetDAO dao;

 public void testCreate() throws Exception{
  IWidget wdgt = new Widget();
  wdgt.setWidgetId(1000);
  wdgt.setPartNumber("12-34-BBD");
  try{
   this.dao.createWidget(wdgt);
  }catch(CreateException e){
   TestCase.fail("CreateException thrown creating a Widget");
  }
 }

 protected void setUp() throws Exception {
  ApplicationContext context =
   new ClassPathXmlApplicationContext("spring-config.xml");
  this.dao = (WidgetDAO) context.getBean("widgetDAO");
 }
}
```

由于 JUnitPerf 是一个基于装饰器的框架，为了真正地驾驭它，必须提供一个 `suite()` 方法并将现有的测试装饰以 `TimedTest`。`TimedTest` 以 `Test` 和执行该测试的最大时间量作为参数。

也可以选择传入一个 `boolean` 标志作为第三个参数（`false`），这将导致测试快速失败 —— 意味着如果超过最大时间，JUnitPerf 将立即 迫使测试失败。否则，测试样例将完整运行，然后失败。区别很微妙：在一个失败的样例中，不带可选标志运行测试可以帮您了解运行总时间。传入 `false` 值却意味着得不到运行总时间。

例如，在清单 2 中，我在运行 `testCreate()` 时设定了一个两秒钟的上限。如果执行总时间超过了这个时间，测试样例将失败。由于我并未传入可选的 `boolean` 参数，该测试将完整运行，而不管运行会持续多久。

清单 **2.** 为生成 **TimedTest** 而实现的 **suite** 方法

```
public static Test suite() {
 long maxElapsedTime = 2000; //2 seconds
 Test timedTest = new TimedTest(
   new WidgetDAOImplTest("testCreate"), maxElapsedTime);
 return timedTest;
}
```

此测试通常在 JUnit 框架中运行 —— 现有的 Ant 任务、Eclipse 运行器等等，会像运行任何其他 JUnit 测试一样运行这个测试。惟一的不同是，该测试将发生在计时器的上下文中。

过度的负载测试

与在测试场景中验证一个方法（或系列方法）的时间限制正好相反，JUnitPerf 也方便了负载测试。正如在 `TimedTest` 中一样，JUnitPerf 的 `LoadTest` 也像装饰器一样运行，它通过将 JUnit `Test` 和额外的线程信息绑定起来，从而模拟负载。

使用 `LoadTest`，可以指定要模拟的用户（线程）数量，甚至为这些线程的启动提供计时机制。JUnitPerf 提供两类 `Timer`：`ConstantTimer` 和 `RandomTimer`。通过为 `LoadTest` 提供这两类计时器，可以更真实地模拟用户负载。如果没有 `Timer`，所有线程都会同时启动。

清单 3 是用 `ConstantTimer` 实现的含 10 个模拟用户的负载测试：

清单 **3.** 为生成负载测试而实现的 **suite** 方法

```
public static Test suite() {
 int users = 10;
 Timer timer = new ConstantTimer(100);
 return new LoadTest(
  new WidgetDAOImplTest("testCreate"),
    users, timer);
}
```

请注意，`testCreate()` 方法运行 10 次，每个线程间隔 100 毫秒启动。未设定时间限制 —— 这些方法完整运行，如果其中任何的方法执行失败，JUnit 会相应地报告失败。

用样式进行装饰

装饰器并不局限于单个的装饰物。例如，在 Java™ I/O 中，可以为 `FileInputStream` 装饰上一个带 `BufferedReader` 的 `InputStreamReader`（只要记住：`BufferedReader in = new BufferedReader(new InputStreamReader(new FileInputStream("infi`）。

装饰可以有多个层次，JUnitPerf 的 `TimedTest` 和 `LoadTest` 也是一样。当这两个类彼此装饰时，将导致一些强制的测试场景，例如像这样的场景：在一项业务中放置了负载并应用了时间限制。或者，我们可以仅仅将之前的两个测试场景以如下方式结合起来：

- 在 `testCreate()` 方法中放置一项负载。
- 规定每个线程必须在该时间限制内结束。

我通过为一个标准 `Test` 装饰上 `LoadTest`（由 `TimedTest` 装饰）应用了上述规范，清单 4 显示了其结果。

清单 **4.** 经装饰的负载和时限测试

```
public static Test suite() {
 int users = 10;
 Timer timer = new ConstantTimer(100);
 long maxElapsedTime = 2000;
 return new TimedTest(new LoadTest(
   new WidgetDAOImplTest("testCreate"), users, timer),
     maxElapsedTime);
}
```

正如您所看到的那样，`testCreate()` 方法运行 10 次（每隔 100 毫秒启动一个线程），且每个线程必须在 2 秒内完成，否则整个测试场景将失败。

使用注意

尽管 JUnitPerf 是一个性能测试框架，但也要先大致估计一下测试要设定的性能指数。这是由于所有由 JUnitPerf 装饰的测试都通过 JUnit 框架运行，所以就存在额外的消耗，特别是在利用 fixture 时。由于 JUnit 本身用一个 `setUp` 和一个 `tearDown()` 方法装饰所有测试样例，所以要在测试场景的整个上下文中考虑执行时间。

相应地，我经常创建使用我想要的 fixture 逻辑的测试，但也会运行一个空白测试来确定性能指数基线。这是一个大致的估计，但它必须作为基线添加到任何想要的测试限制中。

例如，如果运行一个由 fixture 逻辑（使用 DbUnit）装饰的空白测试用时 2.5 秒，那么您想要的所有测试限制都应将这一额外时间考虑在内 —— 这可以从清单 5 中的基准测试中看到：

清单 **5. JUnitPerf** 基准测试

```
public class DBUnitSetUpBenchmarkTest extends DatabaseTestCase {
 private WidgetDAO dao = null;

 public void testNothing(){
  //should be about 2.5 seconds
 }

 protected IDatabaseConnection getConnection() throws Exception {
  Class driverClass = Class.forName("org.hsqldb.jdbcDriver");
  Connection jdbcConnection =
   DriverManager.getConnection(
     "jdbc:hsqldb:hsql://127.0.0.1", "sa", "");
  return new DatabaseConnection(jdbcConnection);
 }

 protected IDataSet getDataSet() throws Exception {
  return new FlatXmlDataSet(new File("test/conf/seed.xml"));
 }

 protected void setUp() throws Exception {
  super.setUp();
  final ApplicationContext context =
   new ClassPathXmlApplicationContext("spring-config.xml");
  this.dao = (WidgetDAO) context.getBean("widgetDAO");
 }
}
```

请注意，清单 5 的测试样例 `testNothing()` 什么都没做。其惟一的目的是确定运行 `setUp()` 方法（当然，该方法也通过 DbUnit 设置了一个数据库）的总时间。

也请记住，测试时间将依赖于机器的配置而变化，同时也依赖于在执行 JUnitPerf 测试时运行的东西而变化。我经常发现，将 JUnitPerf 测试放到它们自己的分类中有助于将它们同标准测试隔离开。这意味着，在运行一个测试时不必每次都运行 JUnitPerf 测试，例如在一个 CI 环境中签入代码。我也会创建特定的 Ant 任务，从而只在精心策划的将性能测试考虑在内的场景或环境中运行这些测试。

试试吧！

用 JUnitPerf 进行性能测试无疑是一门严格的科学，但在开发生命周期的早期，这是确定和监控应用程序代码的低端性能的极佳方式。另外，由于它是一个基于装饰器的 JUnit 扩展框架，所以可以很容易地用 JUnitPerf 装饰现有的 JUnit 测试。

想想您已经花了这么多时间来担心应用程序在负载下会怎样执行。用 JUnitPerf 进行性能测试可以为您减少担忧并节省时间，同时也确保了应用程序代码的质量。

追求代码质量: 通过测试分类实现敏捷构建

以不同频率运行测试来缩短构建持续期

人人都认可开发人员测试的重要性，但为什么运行测试还是需要花费太多时间？本月，Andrew Glover 揭示了三种用来确保端到端系统健壮性的测试类型，随后展示了如何按类型来自动排序及运行测试。即使是使用当今大型测试套件，这样做也能显著地减少构建时间。

如果这样说不会（令您）很痛苦的话，请设想您是一名任职于一家 2002 年早期创建的公司的开发人员。在金钱的驱动下，您和您的团队接到了一项任务，即使用最新且最强大的 Java™ API 构建一个大型的数据驱动的 Web 应用程序。您和公司管理层都坚定不疑地相信这就是最终将被称为敏捷过程 的东西。从第一天起，您就用 JUnit 构建测试，且把它作为 Ant 构建过程的一部分尽可能频繁地运行。还将设置一个定时任务在夜间运行构建。在接下来的某个时刻，有人会下载 CruiseControl，不断增长的测试套件会在每次签入时运行。

时至今日

经过过去几年的努力，您的公司已经开发了一个庞大的代码库和一个同样庞大的 JUnit 测试套件。一切都很正常，直到大约一年前，测试套件包含了 2000 个测试，同时人们开始注意到运行构建过程用时超过三个小时。在此之前的几个月，由于 CI 服务器资源紧张，您在代码签入时通过 Continuous Integration（CI）停止运行单元测试，并将测试切换到夜间运行，这使得之后的早晨时间非常紧张，于是开发人员努力去弄清楚是什么出错以及为什么出错。

这些天，似乎测试套件整晚极少超过一次运行，为什么会这样呢？因为它们费时太多！没人会仅仅为了弄明白系统是否运行良好而几个小时守在那里。此外，整个测试套件都是在晚上运行，不是吗？

由于测试运行得太不频繁，它们常常充满了错误。因而，您和您的团队开始质疑单元测试的价值：如果它们对代码质量那么重要，那又为什么会让人这么头痛呢？你们的结论是：单元测试有其重要的作用，但必须要能用一种更为敏捷的方式运行它们。

尝试测试分类

您所需要的是一个将构建转换到一种更为敏捷状态的策略。您需要这样一种解决方案，使一天当中运行测试的次数超过一次，并使测试套件恢复到要用三个小时才能完 成构建之前的水平。

为完整地恢复整个测试套件，在试图提出一个策略之前，很有必要弄清楚通用术语 “单元测试” 的含义。诸如 “我家有一个动物” 和 “我喜欢车” 这样的表述并不很具体，“我们编写单元测试” 也是一样。这年头，单元测试能代表一切。

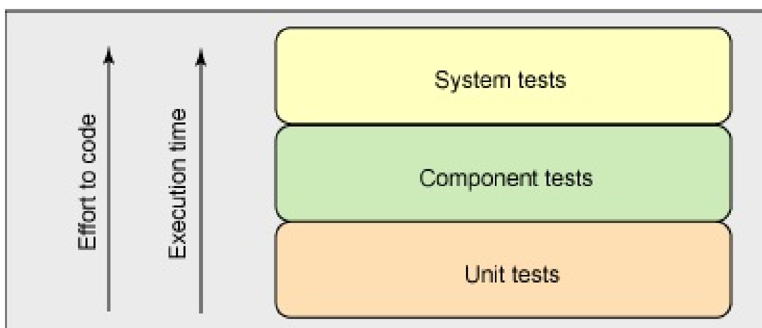
就拿之前有关动物和车的表述来说：它们导致了更多的疑问。例如，您家有哪种动物？是一只猫、一条蜥蜴还是一头熊？“我家有一头熊” 和 “我家有一只猫” 截然不同。同样，当和汽车销售员交谈时，只说 “我喜欢车” 没什么用处。您喜欢哪种车：赛车、卡车还是旅行车？任何一个答案都能带来截然不同的结果。

同样，对于开发人员测试来说，按照类型 将测试分类也是很有用的。这样做能够实现更为精确的语言，并且能使您的团队以不同的频率运行不同的测试类型。为了避免运行所有 “单元测试” 所需的令人恐惧的三小时构建时间，分类是关键。

三种类型

测试套件可以形象地分为三层，每一层代表一种不同的开发人员测试类型，该测试类型由其运行时间的长短决定。正如在图 1 中看到的那样，每一层都增加了总的构建时间，要么增加了运行时间，要么最终增加了编写时间。

图 **1.**测试分类的三个层次



底层由运行时间最短的测试构成，可以想象的到，它们也最易于编写。这些测试占用的代码量也是最少的。顶层由更高级别的测试构成，这些测试占用了应用程序更大的部分。这些测试有一点难于编写，执行时间也要长得多。中间层是处于这两个极端中间的测试类型。

三种类型如下所示：

- 单元测试
- 组件测试
- 系统测试

让我们分别来看一下。

改进代码质量

别错过 Andrew 的相关 讨论论坛，里面有关于代码语法、测试框架以及如何编写专注于质量的代码的帮助。

1. 单元测试

单元测试隔离地验证一个或多个对象。单元测试不处理数据库、文件系统或任何可能延长测试运行时间的内容；因而，从第一天就可以编写单元测试。事实上，这也正是 JUnit 设计的确切目的所在。单元测试的隔离概念有无数的模拟对象库作后盾，这些库便利了将一个特定的对象从其外部依赖项中隔离出来。而且，单元测试能够在真正要测试的代码前编写 —— 由此有了测试优先开发 的概念。

单元测试通常易于编写，因为它们并不依靠于架构的依赖项，且通常运行得很快。缺点是，独立的单元测试只能覆盖稍显有限的代码。单元测试的重大价值在于：它们使开发人员能够在尽可能低的层面上保证对象的可靠性。

由于单元测试运行得如此之快且如此易于编写，代码库中应包含许多单元测试，并且应该尽可能多地运行它们。在执行构建时，应该经常 运行它们，不管是在机器上，还是在 CI 环境的上下文中（这意味着，代码一经签入 SCM 环境，就要运行单元测试）。

---

2. 组件测试

组件测试验证多个相互作用的对象，但它突破了隔离的概念。由于组件测试处理一个架构的多个层次，所以它们经常用于处理数据库、文件系统、网络元素等。同样，提前编写组件测试有点难，所以将其包含至一个真正的测试优先/测试驱动的场景中是很大的挑战。

编写组件测试要花更长的时间，因为它们比单元测试所涉及的东西要多。另一方面，由于其宽广的范围，它们实现了比单元测试更广的代码覆盖率。当然它们也要花更多时间运行，所以同时运行很多的组件测试会显著地 增加总的测试时间。

许多框架有助于测试大型架构组件。DbUnit 是这类框架的一个典型例子。DbUnit 能够很好地处理在测试状态间建立一个数据库这样的复杂性，因而它会使编写依赖于数据库的测试变得较为简单。

当构建的测试延长时，通常都预示着包含了一个大型的组件测试套件。由于这些测试比真正的单元测试运行时间长，因而不能一直运行它们。相应地，在 CI 环境中这些测试可以至少 每小时运行一次。在签入任何代码前，也应该总在一个本地开发人员机器上运行这些测试。

验收测试

验收测试 和功能测试类似，不同之处在于，理想情况下，验收测试是由客户或最终用户编写的。正如功能测试一样，验收测试也像最终用户测试那样进行。Selenium（参见 参考资料）是一个备受瞩目的验收框架，它使用浏览器测试 Web 应用程序。Selenium 在构建过程中可

以是自动运行的，就像 JUnit 测试一样。但 Selenium 是一个新平台：它不使用 JUnit，在使用方式上也不相似。

3. 系统测试

系统测试端到端地 验证一个软件应用程序。因而，它们引入了一个更高级别的架构复杂度：整个应用程序必需为要进行的系统测试而运行。如果是一个 Web 应用程序，您就需要访问数据库以及 Web 服务器、容器和任何与运行系统测试相关的配置。其遵循这样的原则，即大多数系统测试都在软件生命周期的较后周期中编写。

编写系统测试是个挑战，也需要大量的时间来实际地执行。而另一方面，就架构性代码覆盖率来讲，系统测试是一件极为划算的事情。

系统测试和功能测试很相似。所不同的是，它们并不仿效用户，而是模拟出 一个用户。与在组件测试中一样，现在创建了大量的框架来为这些测试提供方便。例如，jWebUnit 通过模拟一个浏览器来测试 Web 应用程序。

用 **jWebUnit** 还是 **Selenium** 呢？

jWebUnit 是为系统测试设计的一个 JUnit 扩展框架；因而它需要您来编写测试。Selenium 在验收测试和功能测试方面表现卓越，不同于 jWebUnit，它使非程序员也能够编写测试。理想情况下，团队可以同时 使用两种工具来验证应用程序的功能。

实现测试分类

所以，您的单元测试套件就是名副其实的包括单元测试、组件测试和系统测试的套件。不仅如此，在检查了这些测试后，您现在知道构建花了三个小时的原因是：绝大部分时间都被组件测试所占用。下一个问题是，如何用 JUnit 实现测试分类？

有几种方式可选，但这里我们只关注于其中两种最简单的方式：

- 根据所需种类创建定制的 JUnit 套件文件。
- 为每种测试类型创建定制目录。

用 **TestNG** 进行测试分类

用 TestNG 实现测试分类相当简单。用 TestNG 的 `group` 注释按照种类在逻辑上划分测试，这与将适当的 `group` 注释应用到所需测试中一样简单。这样一来，运行一个特定类型实际上就是将一个相应的组名称传递给一个测试运行程序，如 Ant。

创建定制套件

可以使用 JUnit 的 `TestSuite` 类（属于 `Test` 类型）来定义许多互相归属的测试。首先，创建一个 `TestSuite` 实例，并为其添加相应的测试类或测试方法。然后，可以通过定义一个叫做 `suite()` 的 `public static` 方法，在 `TestSuite` 实例中指定 JUnit。包含的所有测试随后将在单个运行中执行。因而，可以通过创建单元 `TestSuite`、组件 `TestSuite` 和系统 `TestSuite` 来实现测试分类。

例如，清单 1 中显示的类创建了一个 `TestSuite`，其持有 `suite()` 方法中所有的组件测试。请注意此类并不是非常特定于 JUnit 的。它既没有扩展 `TestCase`，也没有定义任何测试用例。但它会反射性地找到 `suite()` 方法并运行由它返回的所有测试。

清单 **1.** 用于组件测试的 **TestSuite**

```
package test.org.acme.widget;

import junit.framework.Test;
import junit.framework.TestSuite;
import test.org.acme.widget.*;

public class ComponentTestSuite {

 public static void main(String[] args) {
  junit.textui.TestRunner.run(ComponentTestSuite.suite());
 }

 public static Test suite(){
  TestSuite suite = new TestSuite();
  suite.addTestSuite(DefaultSpringWidgetDAOImplTest.class);
  suite.addTestSuite(WidgetDAOImplLoadTest.class);
  ...
  suite.addTestSuite(WidgetReportTest.class);
  return suite;
 }
}
```

定义 `TestSuite` 的过程的确需要浏览现有的测试，并将它们添加到相应的类中（即，将所有的单元测试添加到一个 `UnitTestSuite` 中）。这也意味着，由于在一个给定分类中编写新测试，不得不将它们按照一定的程序添加到适当的 `TestSuite` 中，当然，还需要重新编译 它们。

运行独立的 `TestSuites`，然后试着创建单一的 Ant 任务，Ant 任务调用正确的测试集。可以定义一个 `component-test` 任务，用于组织 `ComponentTestSuite` 等，正如清单 2 中所示：

清单 **2.** 只运行组件测试的 **Ant** 任务

```
<target name="component-test"
           if="Junit.present"
           depends="junit-present,compile-tests">
 <mkdir dir="${testreportdir}"/>
 <junit dir="./" failureproperty="test.failure"
             printSummary="yes"
             fork="true" haltonerror="true">
   <sysproperty key="basedir" value="."/>
   <formatter type="xml"/>
   <formatter usefile="false" type="plain"/>
   <classpath>
    <path refid="build.classpath"/>
    <pathelement path="${testclassesdir}"/>
    <pathelement path="${classesdir}"/>
   </classpath>
   <batchtest todir="${testreportdir}">
    <fileset dir="test">
     <include name="**/ComponentTestSuite.java"/>
    </fileset>
   </batchtest>
 </junit>
</target>
```

理想情况下，还需要有调用单元测试和系统测试的任务。最后，在想要运行整个测试套件时，应该创建一个依赖于所有三种测试种类的第四项任务，如清单 3 中如示：

清单 **3.** 用于所有测试的 **Ant** 任务

```
<target name="test-all" depends="unit-test,component-test,system-test"/>
```

创建定制 `TestSuite` 是实现测试分类的一个快速解决方案。这个方法的缺点是：一旦创建新测试，就必须通过编程将它们添加到适当的 `TestSuite` 中，这很痛苦。为每种测试创建定制目录更具扩展性，且允许不 经过重新编译就添加新的经过分类的测试。

创建定制目录

我发现，用 JUnit 实现测试分类最简单的方法是将测试在逻辑上划分为与其测试类型相应的特定目录。使用这项技术，所有的单元测试将驻留在一个 *unit* 目录中，所有的组件测试将驻留在一个 *component* 目录中，依此类推。

例如，在一个保存所有未分类测试的 *test* 目录中，可以创建三个新的子目录，如清单 4 所示：

清单 **4.** 实现测试分类的目录结构

```
acme-proj/
       test/
          unit/
          component/
          system/
          conf/
```

为运行这些测试，必需至少定义四个 Ant 任务：为单元测试定义一个，为组件测试定义一个，依此类推。第 4 项任务是一个方便的任务，它运行所有三种测试类型（如 清单 3 所示）。

该 JUnit 任务和 清单 2 中定义的任务非常相似。所不同的是该任务 `batchtest` 方面的一个细节。此时，`fileset` 指向一个具体的目录。在清单 5 的例子中，它指向 *unit* 目录。

清单 **5.** 用于运行所有单元测试的 **JUnit** 任务的批量测试方面

```
<batchtest todir="${testreportdir}">
 <fileset dir="test/unit">
  <include name="**/**Test.java"/>
 </fileset>
</batchtest>
```

请注意，这个测试只运行 *test/unit* 目录下的所有测试。当创建了新的单元测试（或针对此问题的任何其他测试），只需要将它们放到该目录下，一切就准备妥当了！比起需要将一行新代码添加到 `TestSuite` 文件并进行重新编译，这样还是多少简单了一点。

问题解决了！

回到最初的场景中，假设您和您的团队认为使用特定目录是针对构建时间问题的最具扩展性的解决方案。该任务最困难的地方是检查及分配测试类型。您重构了 Ant 构建文件并创建了 4 项新任务（为单个的测试类型创建了三项，为运行所有这些测试类型创建了一项）。不仅如此，您还修改了 CruiseControl，从而只在（代码）签入时运行真正的单元测试，并以小时为基础运行组件测试。在进一步检查之后，发现系统测试也可以按小时运行，所以您创建了一个将组件测试和系统测试一起运行的额外任务。

最终结果是，测试每天都运行很多次，您的团队能够更快地发现集成错误 —— 通常在几个小时之内。

当然，创建敏捷性构建并未解决全部问题，但它在确保代码质量方面确实扮演了至关重要的角色。测试运行得更加频繁了，针对开发人员测试价值的顾虑成为一段遥远的记忆。另外，更重要的是，现在 2006 年您的公司获得了极大的成功！

追求代码质量: 可重复的系统测试

用 *Cargo* 进行自动容器管理

在测试加入到 servlet 容器的 Web 应用程序时，编写符合逻辑的可重复的测试尤其需要技巧。在 Andrew Glover 的提高代码质量的这个续篇中，他介绍了 Cargo，这是一个以通用方式自动化容器管理的开源框架，有了这个框架，您可以随时编写符合逻辑的可重复的系统测试。

在本质上，像 JUnit 和 TestNG 一样的测试框架方便了可重复性测试的创建。由于这些框架利用了简单 Boolean 逻辑（以 *assert* 方法的形式）的可靠性，这使得无人为干预而运行测试成为可能。事实上，自动化是测试框架的主要优点之一 —— 我能够编写一个用于断言具体行为的相当复杂的测试，且一旦这些行为有所改变，框架就会报告一个人人都能明白的错误。

利用成熟的测试框架会带来框架 可重复性的优点，这是显而易见的。但逻辑的 可重复性却取决于您。例如，考虑创建用于验证 Web 应用程序的可重复测试的情况，一些 JUnit 扩展框架（如 JWebUnit 和 HttpUnit）在协助自动化的 Web 测试方面非常好用。但是，使测试的 *plumbing* 可重复则是开发人员的任务，而这在部署 Web 应用程序资源时很难进行。

实际的 JWebUnit 测试的构造过程相当简单，如清单 1 所示：

清单 **1.** 一个简单的 **JWebUnit** 测试

```
package test.come.acme.widget.Web;

import net.sourceforge.jwebunit.WebTester;
import junit.framework.TestCase;

public class WidgetCreationTest extends TestCase {
 private WebTester tester;

 protected void setUp() throws Exception {
  this.tester = new WebTester();
  this.tester.getTestContext().
   setBaseUrl("http://localhost:8080/widget/");
 }

 public void testWidgetCreation() {
  this.tester.beginAt("/CreateWidget.html");
  this.tester.setFormElement("widget-id", "893-44");
  this.tester.setFormElement("part-num", "rt45-3");

  this.tester.submit();
  this.tester.assertTextPresent("893-44");
  this.tester.assertTextPresent("successfully created.");
 }
}
```

这个测试与一个 Web 应用程序通信，并试图创建一个基于该交互的小部件。该测试随后校验此部件是否被成功创建。读过本系列之前部分的读者们也许会注意到该测试的一个微妙的可重复性问题。您注意到了吗？如果这个测试用例连续 运行两次会怎样呢？

改进代码质量

不要错过了 Andrew 的附随的 讨论论坛，获取有关代码编写方法、测试框架及编写高质量代码的帮助。

由这个小部件实例（即，`widget-id`）的验证方面可以判断出，可以安全地做出这样的假设，即此应用程序中的数据库约束很可能会阻止创建一个已经存在的额外的小部件。由于缺少了一个在运行另一个测试前删除此测试用例的目标小部件的过程，如果再连续运行两次，这个测试用例非常有可能会失败。

幸运的是，如前面文章中所探讨的那样，有一个有助于数据库-依赖性（database-dependent）测试用例可重复性的机制 —— 即 DbUnit。

使用 DbUnit

改进 清单 1 中的测试用例来使用 DbUnit 是非常简单的。DbUnit 只需要一些插入数据库的数据和一个相应的数据库连接，如清单 2 所示：

清单 **2.** 用 **DbUnit** 进行的数据库**-**依赖性测试

```
package test.come.acme.widget.Web;

import java.io.File;
import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

import org.dbunit.database.DatabaseConnection;
import org.dbunit.database.IDatabaseConnection;
import org.dbunit.dataset.DataSetException;
import org.dbunit.dataset.IDataSet;
import org.dbunit.dataset.xml.FlatXmlDataSet;
import org.dbunit.operation.DatabaseOperation;

import net.sourceforge.jwebunit.WebTester;
import junit.framework.TestCase;

public class RepeatableWidgetCreationTest extends TestCase {
 private WebTester tester;

 protected void setUp() throws Exception {
  this.handleSetUpOperation();
  this.tester = new WebTester();
  this.tester.getTestContext().
   setBaseUrl("http://localhost:8080/widget/");
 }

 public void testWidgetCreation() {
  this.tester.beginAt("/CreateWord.html");
  this.tester.setFormElement("widget-id", "893-44");
  this.tester.setFormElement("part-num", "rt45-3");

  this.tester.submit();
  this.tester.assertTextPresent("893-44");
  this.tester.assertTextPresent("successfully created.");
 }

 private void handleSetUpOperation() throws Exception{
  final IDatabaseConnection conn = this.getConnection();
  final IDataSet data = this.getDataSet();
  try{
   DatabaseOperation.CLEAN_INSERT.execute(conn, data);
  }finally{
   conn.close();
  }
 }

 private IDataSet getDataSet() throws IOException, DataSetException {
  return new FlatXmlDataSet(new File("test/conf/seed.xml"));
 }

 private IDatabaseConnection getConnection() throws
   ClassNotFoundException, SQLException {
    Class.forName("org.hsqldb.jdbcDriver");
    final Connection jdbcConnection =
     DriverManager.getConnection("jdbc:hsqldb:hsql://127.0.0.1",
      "sa", "");
    return new DatabaseConnection(jdbcConnection);
 }
}
```

加入了 DbUnit，测试用例真的是可重复的了。在 `handleSetUpOperation()` 方法中，每当运行一个测试用例时，DbUnit 对数据执行一个 `CLEAN_INSERT` 。此操作本质上将一个数据库的数据清空并插入一个新的数据集，从而删除任何之前创建的小部件。

再一次探讨什么是 DbUnit？

DbUnit 是一个 JUnit 扩展，用于在运行测试时将数据库放入一个已知状态中。开发人员使用 XML 种子文件将特定数据插入到测试用例所依赖的数据库中。因而，DbUnit 便利了依赖于一个或多个数据库的测试用例的可重复性。

但那并不意味着已经结束了对测试用例可重复性这一话题的探讨。事实上，一切才刚刚开始。

重复系统测试

我喜欢将 清单 1 和 清单 2 中定义的测试用例称为系统测试。因为系统测试运行安装完整的应用程序，如 Web 应用程序，它们通常包含一个 servlet 容器和一个相关联的数据库。这些测试的目的在于校验那些设计为端对端操作的外部接口（如 Web 应用程序中的 Web 页面）。

弹性优先级

作为总体规则，应在任何可能的时候避免测试用例继承。许多 JUnit 扩展框架都提供特定的可继承测试用例，以便利于测试一个特定的架构。然而由于 Java™ 平台的单一继承范例，使得从框架中继承类的测试用例饱受缺乏弹性之苦。通常，这些相同的 JUnit 扩展框架提供了代理 API，这使得联合各种不具有严格继承结构的框架变得十分简单。

由于设计它们的目的是为了测试功能完整的应用程序，因而系统测试趋向于增加运行次数而不是减少设置测试的总时间。例如，清单 1 和 清单 2 中展示的逻辑测试在运行前 需要下列步骤：

1. 创建一个 war 文件，该文件包含所有相关 Web 内容，如 JSP 文件、servlet、第三方的 jar 文件、图像等。
2. 将此 war 文件部署到目标 Web 容器中。（如果该容器尚未启动，启动该容器。）
3. 启动任何相关的数据库。（如果需要更新数据库模式，在启动前进行更新。）

现在，对于一个微不足道的小测试要做大量的辅助性工作！如果证明这个过程是耗时的，那么您认为这个测试会间隔多长时间运行一次呢？面对要使系统测试在逻辑上可重复（在一个连续的集成环境中）这一需求，这个步骤列表的确令人望而生畏。

介绍 Cargo

好消息是可以在之前的列表中使所有主要设置步骤自动化。事实上，如果恰好从事过 Java Web 开发，可能已经用 Ant、Maven 或其他构建工具使步骤 1 自动化了。

步骤 2 却是一个有趣的障碍。自动化一个 Web 容器还是需要一定技巧的。例如，一些容器具有定制的 Ant 任务，这些任务方便了其自动部署及运行，但这些任务是特定于容器的。而且，这些任务还有一些假设，如容器的安装位置，还有更重要的是，容器已被安装。

Cargo 是一个致力于以通用方式自动化容器管理的创新型开源项目，因而用于将 WAR 文件部署到 JBoss 的相同的 API 也能够启动及停止 Jetty。Cargo 也能自动下载并安装一个容器。可以以不同的方式利用 Cargo 的 API，从 Java 代码到 Ant 任务，再到 Maven 目标。

运用一个如 Cargo 这样的工具，应对了在编写合乎逻辑可重复的测试用例中遇到的主要问题之一。另外，还可以构造一个构建用于驾驭 Cargo 的功能以 自动地完成下列任务：

1. 下载一个所期望的容器。
2. 安装该容器。
3. 启动该容器。
4. 将一个选定的 WAR 或 EAR 文件部署到该容器中。

很简单，是吧？接下来，您还能够用 Cargo 停止一个选定的容器。

“谈谈” Cargo

在深入 Cargo 前，最好先了解一下 Cargo 的基础知识。也就是说，由于 Cargo 与容器及容器管理相关，所以要理解了容器及容器管理的有关概念。

对于新手，显然要先了解容器 的概念。容器是用以寄存应用程序的服务器。应用程序可以是基于 Web 的，基于 EJB 的，或基于这两者的，这就是为什么有 Web 容器和 EJB 容器的原因。Tomcat 是 Web 容器，而 JBoss 则会被认为是 EJB 容器。因此，Cargo 支持相当多的容器，但在我的例子中，我将使用 Tomcat 5.0.28 版。（Cargo 将称其为“tomcat5x”容器。）

接下来，如果尚未安装容器，可以使用 Cargo 来下载并安装一个特定的容器。为此，需要提供给 Cargo 一个下载 URL。一旦安装了容器，Cargo 也会允许使用配置选项 来对其进行配置。这些选项以名称-值对的形式存在。

最后，要介绍可部署资源 的概念，在我的例子中即 WAR 文件。请注意 EAR 文件也是一样的简单。

将这些概念记住，让我们来看一下可以用 Cargo 来完成什么任务。

Cargo 实践

本文中的例子涉及到在 Ant 中使用 Cargo，这就必需将之前定义的系统测试和 Cargo Ant 任务包装在一起。这些任务随后安装、启动、部署并停止容器。我们将首先进行安装设置，运行测试然后停止容器。

在 Ant 构建中使用 Cargo 所需的第一步是提供一个针对所有的 Cargo 任务的任务定义。这一步允许随后在构建文件中引用 Cargo 任务。应付这一步有很多的方法。清单 3 简单地装载了来自 Cargo JAR 文件中的属性文件的任务：

清单 **3.** 在 **Ant** 中装载所有的 **Cargo** 任务

```
<taskdef resource="cargo.tasks">
 <classpath>
  <pathelement location="${libdir}/${cargo-jar}"/>
  <pathelement location="${libdir}/${cargo-ant-jar}"/>
 </classpath>
</taskdef>
```

一但定义了 Cargo 的任务，真正的行动就开始了。清单 4 定义了下载、安装及启动 Tomcat 容器的 Cargo 任务。`zipurlinstaller` 任务将 Tomcat 从 `http://www.apache.org/dist/tomcat/tomcat-5/v5.0.28/bin/ jakarta-tomcat-5.0.28.zip` 中下载并安装到一个本地临时目录中。

清单 **4.** 下载并启动 **Tomcat 5.0.28**

```
<cargo containerId="tomcat5x" action="start"
       wait="false" id="${tomcat-refid}">

 <zipurlinstaller installurl="${tomcat-installer-url}"/>

 <configuration type="standalone" home="${tomcatdir}">
  <property name="cargo.remote.username" value="admin"/>
  <property name="cargo.remote.password" value=""/>

  <deployable type="war" file="${wardir}/${warfile}"/>

 </configuration>

</cargo>
```

请注意要想如您所愿，从不同的任务中启动和停止一个容器，必需将容器同一个惟一的 id 联系起来，此 id 是 `cargo` 任务的 `id="${tomcat-refid}"` 。

还要注意的是，Tomcat 的配置是在 `cargo` 任务内处理的。在 Tomcat 中，必需设置 `username` 和 `password` 属性。最后，使用 `deployable` 元素定义一个指向 WAR 文件的指针。

Cargo 属性

Cargo 任务中用到的所有属性都显示在清单 5 中。例如，`tomcatdir` 定义 Tomcat 将安装的两个位置中的一个。这个特别的位置是一个镜像结构，该位置将被实际下载并安装的 Tomcat 实例（在临时目录中找到的）所引用。`tomcat-refid` 属性则帮助将容器中惟一的实例与其镜像关联起来。

清单 **5. Cargo** 属性

```
<property name="tomcat-installer-url"
  value="http://www.apache.org/dist/tomcat/tomcat-5/v5.0.28/bin/
    jakarta-tomcat-5.0.28.zip"/>
<property name="tomcatdir" value="target/tomcat"/>
<property name="tomcat.username" value="admin"/>
<property name="tomcat.passwrd" value=""/>
<property name="wardir" value="target/war"/>
<property name="warfile" value="words.war"/>
<property name="tomcat-refid" value="tmptmct01"/>
```

为停止一个容器，可以定义一个引用 `tomcat-refid` 属性的任务，如清单 6 所示。

清单 **6.** 按 **Cargo** 方式停止容器

```
<cargo containerId="tomcat5x" action="stop"
       refid="${tomcat-refid}"/>
```

用 Cargo 封装

清单 7 将 清单 4 和清单 6 中的代码联合起来，用两个 Cargo 任务封装了一个测试目标：一个用于启动 Tomcat，另一个用于停止 Tomcat。`antcall` 任务调用在清单 8 中定义的名为 `_run-system-tests` 的目标。

清单 **7.** 用 **Cargo** 封装测试目标

```
<target name="system-test" if="Junit.present"
        depends="init,junit-present,compile-tests,war">

 <cargo containerId="tomcat5x" action="start"
       wait="false" id="${tomcat-refid}">
  <zipurlinstaller installurl="${tomcat-installer-url}"/>
  <configuration type="standalone" home="${tomcatdir}">
   <property name="cargo.remote.username" value="admin"/>
   <property name="cargo.remote.password" value=""/>
   <deployable type="war" file="${wardir}/${warfile}"/>
  </configuration>
 </cargo>

 <antcall target="_run-system-tests"/>

 <cargo containerId="tomcat5x" action="stop"
       refid="${tomcat-refid}"/>

</target>
```

清单 8 定义测试目标，称作 `_run-system-tests`。请注意此任务只 运行置于 `test/system` 目录下的系统测试。例如，清单 2 中定义的测试用例就位于这个目录下。

清单 **8.** 通过 **Ant** 运行 **JUnit**

```
<target name="_run-system-tests">
 <mkdir dir="${testreportdir}"/>
 <junit dir="./" failureproperty="test.failure"
        printSummary="yes" fork="true"
        haltonerror="true">
  <sysproperty key="basedir" value="."/>
  <formatter type="xml"/>
  <formatter usefile="false" type="plain"/>
  <classpath>
   <path refid="build.classpath"/>
   <pathelement path="${testclassesdir}"/>
   <pathelement path="${classesdir}"/>
  </classpath>
  <batchtest todir="${testreportdir}">
   <fileset dir="test/system">
    <include name="**/**Test.java"/>
   </fileset>
  </batchtest>
 </junit>
</target>
```

在 清单 7 中，完整地配置了 Ant 构建文件，从而将系统测试与 Cargo 部署封装在一起。清单 7 中的代码确保了清单 8 中 `test/system` 目录下的所有系统测试都是逻辑上可重复的。可以在任何时间里在任何机器上运行这些系统测试，对于连续集成环境尤佳。该测试对容器未做任何假设 —— 未对位置做假设，甚至未对其是否运行做假设！（当然，这些测试仍做了一个假设，我没有强调，即潜在的数据库是配置良好且在运行中的。但那又是另一个要讨论的主题了。）

可重复的结果

在清单 9 中，可以看到工作的成果。当将 `system-test` 命令发布到 Ant 构建后，就会执行系统测试。Cargo 处理管理所选容器的所有细节，不需要对测试环境作出绝对重复性假设。

清单 **9.** 增强的构建

```
war:
 [war] Building war: C:\dev\projects\acme\target\widget.war

system-test:

_run-system-tests:
 [mkdir] Created dir: C:\dev\projects\acme\target\test-reports
 [junit] Running test.come.acme.widget.Web.RepeatableWordCreationTest
 [junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 4.53 sec
 [junit] Testcase: testWordCreation took 4.436 sec

BUILD SUCCESSFUL
Total time: 1 minute 2 seconds
```

请记住，Cargo 也在 Maven 构建中起作用。另外，从正常的应用程序到测试用例，Cargo Java API 都有助于容器的程序化管理。且 Cargo 不仅适用于 JUnit（尽管样例代码是用 JUnit 写的），TestNG 用户将会很高兴地了解到 Cargo 对其测试套件也起作用。事实上，测试用什

么编写并不重要，重要的是将它们同 Cargo 封装起来，容器管理问题就会迎刃而解！

结束语

您的测试是否在逻辑上可重复由您来决定，但是通过本文您确实看到 Cargo 的确很有用处。Cargo 管理容器环境，所以您就可以不用管理。将 Cargo 包含到您的测试例程中 —— 这毫无疑问会减轻您构造用于验证 Web 应用程序的可重复测试的负担。

追求代码质量: JUnit 4 与 TestNG 的对比

为什么 *TestNG* 框架依然是大规模测试的较好选择？

JUnit 4 具有基于注释的新框架，它包含了 TestNG 一些最优异的特性。但这是否意味着 JUnit 4 已经淘汰了 TestNG？Andrew Glover 探讨了这两种框架各自的独特之处，并阐述了 TestNG 独有的三种高级测试特性。

经过长时间积极的开发之后，JUnit 4.0 于今年年初发布了。JUnit 框架的某些最有趣的更改 —— 特别是对于本专栏的读者来说 —— 正是通过巧妙地使用注释实现的。除外观和风格方面的显著改进外，新框架的特性使测试用例的编制从结构规则中解放出来。使原来僵化的 fixture 模型更为灵活，有利于采取可配置程度更高的方法。因此，JUnit 框架不再强求把每一项测试工作定义为一个名称以 `test` 开始的方法，并且现在可以只运行一次 fixture，而不是每次测试都需要运行一次。

虽然这些改变令人欣慰，但 JUnit 4 并不是第一个提供基于注释的灵活模型的 Java™ 测试框架。在修改 JUnit 之前很久，TestNG 就已建立为一个基于注释的框架。

事实上，是 TestNG 在 Java 编程中率先 实现了利用注释进行测试，这使它成为 JUnit 的有力竞争对手。然而，自从 JUnit 4 发布后，很多开发者质疑：二者之间还有什么差别吗？在本月的专栏中，我将讨论 TestNG 不同于 JUnit 4 的一些特性，并提议采用一些方法，使得这两个框架能继续互相补充，而不是互相竞争。

您知道吗？

在 Ant 中运行 JUnit 4 测试比预计的要难得多。事实上，一些团队已发现，惟一的解决方法是升级到 Ant 1.7。

表面上的相似

JUnit 4 和 TestNG 有一些共同的重要特性。这两个框架都让测试工作简单得令人吃惊（和愉快），给测试工作带来了便利。二者也都拥有活跃的社区，为主动开发提供支持，同时生成丰富的文档。

提高代码质量

要找到您最迫切问题的答案，请不要错过 Andrew 的 论坛。

两个框架的不同在于核心设计。JUnit 一直 是一个单元测试框架，也就是说，其构建目的是促进单个对象的测试，它确实能够极其有效地完成此类任务。而 TestNG 则是用来解决更高 级别的测试问题，因此，它具有 JUnit 中所没有的一些特性。

一个简单的测试用例

初看起来，JUnit 4 和 TestNG 中实现的测试非常相似。为了更好地理解我的意思，请看一下清单 1 中的代码。这是一个 JUnit 4 测试，它有一个 macro-fixture（即仅在所有测试运行前调用一次的 fixture），这个 macro-fixture 由 `@BeforeClass` 属性表示：

清单 **1.** 一个简单的 **JUnit 4** 测试用例

```
package test.com.acme.dona.dep;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import org.junit.BeforeClass;
import org.junit.Test;

public class DependencyFinderTest {
 private static DependencyFinder finder;

 @BeforeClass
 public static void init() throws Exception {
  finder = new DependencyFinder();
 }

 @Test
 public void verifyDependencies()
  throws Exception {
   String targetClss =
     "test.com.acme.dona.dep.DependencyFind";

   Filter[] filtr = new Filter[] {
      new RegexPackageFilter("java|junit|org")};

   Dependency[] deps =
      finder.findDependencies(targetClss, filtr);

   assertNotNull("deps was null", deps);
   assertEquals("should be 5 large", 5, deps.length);
 }
}
```

JUnit 用户会立即注意到：这个类中没有了以前版本的 JUnit 中所要求的一些语法成分。这个类没有 `setUp()` 方法，也不对 `TestCase` 类进行扩展，甚至也没有哪个方法的名称以 `test` 开始。这个类还利用了 Java 5 的一些特性，例如静态导入，很明显地，它还使用了注释。

更多的灵活性

在清单 2 中，您可以看到同一个 测试项目。不过这次是用 TestNG 实现的。这里的代码跟清单 1 中的测试代码有个微妙的差别。发现了吗？

清单 **2.** 一个 **TestNG** 测试用例

```
package test.com.acme.dona.dep;

import static org.testng.Assert.assertEquals;
import static org.testng.Assert.assertNotNull;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.Configuration;
import org.testng.annotations.Test;

public class DependencyFinderTest {
 private DependencyFinder finder;

 @BeforeClass
 private void init(){
  this.finder = new DependencyFinder();
 }

 @Test
 public void verifyDependencies()
  throws Exception {
   String targetClss =
     "test.com.acme.dona.dep.DependencyFind";

   Filter[] filtr = new Filter[] {
      new RegexPackageFilter("java|junit|org")};

   Dependency[] deps =
      finder.findDependencies(targetClss, filtr);

   assertNotNull(deps, "deps was null" );
   assertEquals(5, deps.length, "should be 5 large");
 }
}
```

显然，这两个清单很相似。不过，如果仔细看，您会发现 TestNG 的编码规则比 JUnit 4 更灵活。清单 1 里，在 JUnit 中我必须把 `@BeforeClass` 修饰的方法声明为 `static`，这又要求我把 fixture，即 `finder` 声明为 `static`。我还必须把 `init()` 声明为 `public`。看看清单 2，您就会发现不同。这里不再需要那些规则了。我的 `init()` 方法既不是 `static`，也不是 `public`。

从最初起，TestNG 的灵活性就是其主要优势之一，但这并非它惟一的卖点。TestNG 还提供了 JUnit 4 所不具备的其他一些特性。

依赖性测试

JUnit 框架想达到的一个目标就是测试隔离。它的缺点是：人们很难确定测试用例执行的顺序，而这对于任何类型的依赖性测试都非常重要。开发者们使用了多种技术来解决这个问题，例如，按字母顺序指定测试用例，或是更多地依靠 fixture 来适当地解决问题。

如果测试成功，这些解决方法都没什么问题。但是，如果测试不成功，就会产生一个很麻烦的后果：所有 后续的依赖测试也会失败。在某些情况下，这会使大型测试套件报告出许多不必要的错误。例如，假设有一个测试套件测试一个需要登录的 Web 应用程序。您可以创建一

个有依赖关系的方法，通过登录到这个应用程序来创建整个测试套件，从而避免 JUnit 的隔离机制。这种解决方法不错，但是如果登录失败，即使登录该应用程序后的其他功能都正常工作，整个测试套件依然会全部失败！

跳过，而不是标为失败

与 JUnit 不同，TestNG 利用 `Test` 注释的 `dependsOnMethods` 属性来应对测试的依赖性问题。有了这个便利的特性，就可以轻松指定依赖方法。例如，前面所说的登录将在某个方法之前 运行。此外，如果依赖方法失败，它将被跳过，而不是标记为失败。

清单 **3.** 使用 **TestNG** 进行依赖性测试

```
import net.sourceforge.jwebunit.WebTester;

public class AccountHistoryTest  {
 private WebTester tester;

 @BeforeClass
 protected void init() throws Exception {
  this.tester = new WebTester();
  this.tester.getTestContext().
   setBaseUrl("http://div.acme.com:8185/ceg/");
 }

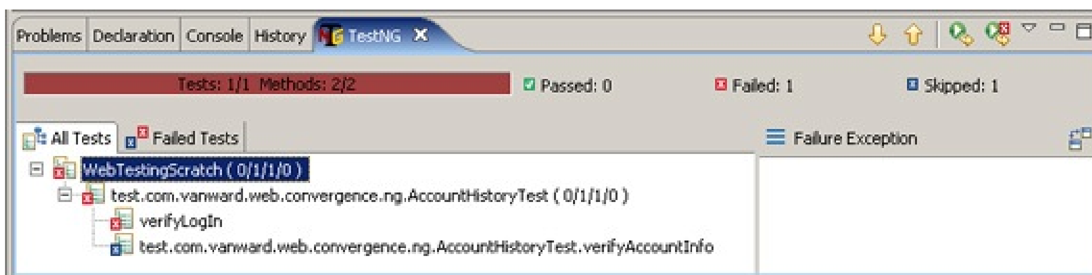
 @Test
 public void verifyLogIn() {
  this.tester.beginAt("/");
  this.tester.setFormElement("username", "admin");
  this.tester.setFormElement("password", "admin");
  this.tester.submit();
  this.tester.assertTextPresent("Logged in as admin");
 }

 @Test (dependsOnMethods = {"verifyLogIn"})
 public void verifyAccountInfo() {
  this.tester.clickLinkWithText("History", 0);
  this.tester.assertTextPresent("GTG Data Feed");
 }
}
```

在清单 3 中定义了两个测试：一个验证登录，另一个验证账户信息。请注意，通过使用 `Test` 注释的 `dependsOnMethods = {"verifyLogIn"}` 子句，`verifyAccountInfo` 测试指定了它依赖 `verifyLogIn()` 方法。

通过 TestNG 的 Eclipse 插件（例如）运行该测试时，如果 `verifyLogIn` 测试失败，TestNG 将直接跳过 `verifyAccountInfo` 测试，请参见图 1：

图 **1.** 在 **TestNG** 中跳过的测试



对于大型测试套件，TestNG 这种不标记为失败，而只是跳过的处理方法可以减轻很多压力。您的团队可以集中精力查找为什么百分之五十的测试套件被跳过，而不是去找百分之五十的测试套件失败的原因！更有利的是，TestNG 采取了只重新运行失败测试的机制，这使它的依赖性测试设置更为完善。

失败和重运行

在大型测试套件中，这种重新运行失败测试的能力显得尤为方便。这是 TestNG 独有的一个特性。在 JUnit 4 中，如果测试套件包括 1000 项测试，其中 3 项失败，很可能就会迫使您重新运行整个测试套件（修改错误以后）。不用说，这样的工作可能会耗费几个小时。

一旦 TestNG 中出现失败，它就会创建一个 XML 配置文件，对失败的测试加以说明。如果利用这个文件执行 TestNG 运行程序，TestNG 就只 运行失败的测试。所以，在前面的例子里，您只需重新运行那三个失败的测试，而不是整个测试套件。

实际上，您可以通过清单 2 中的 Web 测试的例子自己看到这点。`verifyLogIn()` 方法失败时，TestNG 自动创建一个 testng-failed.xml 文件。该文件将成为如清单 4 所示的替代性测试套件：

清单 **4.** 失败测试的 **XML** 文件

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite thread-count="5" verbose="1" name="Failed suite [HistoryTesting]"
       parallel="false" annotations="JDK5">
 <test name="test.com.acme.ceg.AccountHistoryTest(failed)" junit="false">
  <classes>
   <class name="test.com.acme.ceg.AccountHistoryTest">
    <methods>
     <include name="verifyLogIn"/>
    </methods>
   </class>
  </classes>
 </test>
</suite>
```

运行小的测试套件时，这个特性似乎没什么大不了。但是如果您的测试套件规模较大，您很快就会体会到它的好处。

参数化测试

TestNG 中另一个有趣的特性是参数化测试。在 JUnit 中，如果您想改变某个受测方法的参数组，就只能给每个 不同的参数组编写一个测试用例。多数情况下，这不会带来太多麻烦。然而，我们有时会碰到一些情况，对其中的业务逻辑，需要运行的测试数目变化范围很大。

在这样的情况下，使用 JUnit 的测试人员往往会转而使用 FIT 这样的框架，因为这样就可以用表格数据驱动测试。但是 TestNG 提供了开箱即用的类似特性。通过在 TestNG 的 XML 配置文件中放入参数化数据，就可以对不同的数据集重用同一个测试用例，甚至有可能会得到不同的结果。这种技术完美地避免了只能 假定一切正常的测试，或是没有对边界进行有效验证的情况。

在清单 5 中，我用 Java 1.4 定义了一个 TestNG 测试，该测试可接收两个参数：`classname` 和 `size`。这两个参数可以验证某个类的层次结构（也就是说，如果传入 `java.util.Vector`，则 `HierarchyBuilder` 所构建的 `Hierarchy` 的值将为 `2`）。

清单 **5.** 一个 **TestNG** 参数化测试

```
package test.com.acme.da;

import com.acme.da.hierarchy.Hierarchy;
import com.acme.da.hierarchy.HierarchyBuilder;

public class HierarchyTest {
 /**
  * @testng.test
  * @testng.parameters value="class_name, size"
  */
 public void assertValues(String classname, int size) throws Exception{
  Hierarchy hier = HierarchyBuilder.buildHierarchy(classname);
  assert hier.getHierarchyClassNames().length == size: "didn't equal!";
 }
}
```

清单 5 列出了一个泛型测试，它可以采用不同的数据反复重用。请花点时间思考一下这个问题。如果有 10 个不同的参数组合需要在 JUnit 中测试，您只能写 10 个测试用例。每个测试用例完成的任务基本是相同的，只是受测方法的参数有所改变。但是，如果使用参数化测试，就可以只定义一个 测试用例，然后，（举例来说）把所需的参数模式加到 TestNG 的测试套件文件中。清单 6 中展示了这中方法：

清单 **6.** 一个 **TestNG** 参数化测试套件文件

```
<!DOCTYPE suite SYSTEM "http://beust.com/testng/testng-1.0.dtd">
<suite name="Deckt-10">
 <test name="Deckt-10-test">

  <parameter name="class_name" value="java.util.Vector"/>
  <parameter name="size" value="2"/>

  <classes>
   <class name="test.com.acme.da.HierarchyTest"/>
  </classes>
 </test>
</suite>
```

清单 6 中的 TestNG 测试套件文件只对该测试定义了一个参数组（`class_name` 为 `java.util.Vector`，且 `size` 等于 `2`），但却具有无限的可能。这样做的一个额外的好处是：将测试数据移动到 XML 文件的无代码工件就意味着非程序员也可以指定数据。

高级参数化测试

尽管从一个 XML 文件中抽取数据会很方便，但偶尔会有些测试需要有复杂类型，这些类型无法用 `String` 或原语值来表示。TestNG 可以通过它的 `@DataProvider` 注释处理这样的情况。`@DataProvider` 注释可以方便地把复杂参数类型映射到某个测试方法。例如，清单 7 中的 `verifyHierarchy` 测试中，我采用了重载的 `buildHierarchy` 方法，它可接收一个 `Class` 类型的数据, 它断言（asserting）`Hierarchy` 的 `getHierarchyClassNames()` 方法应该返回一个适当的字符串数组：

清单 7. TestNG 中的 DataProvider 用法

```
package test.com.acme.da.ng;

import java.util.Vector;

import static org.testng.Assert.assertEquals;
import org.testng.annotations.DataProvider;
import org.testng.annotations.Test;

import com.acme.da.hierarchy.Hierarchy;
import com.acme.da.hierarchy.HierarchyBuilder;

public class HierarchyTest {

 @DataProvider(name = "class-hierarchies")
 public Object[][] dataValues(){
  return new Object[][]{
   {Vector.class, new String[] {"java.util.AbstractList",
     "java.util.AbstractCollection"}},
   {String.class, new String[] {}}
  };
 }

 @Test(dataProvider = "class-hierarchies")
 public void verifyHierarchy(Class clzz, String[] names)
  throws Exception{
    Hierarchy hier = HierarchyBuilder.buildHierarchy(clzz);
    assertEquals(hier.getHierarchyClassNames(), names,
      "values were not equal");
 }
}
```

`dataValues()` 方法通过一个多维数组提供与 `verifyHierarchy` 测试方法的参数值匹配的数据值。TestNG 遍历这些数据值，并根据数据值调用了两次 `verifyHierarchy`。在第一次调用时，`Class` 参数被设置为 `Vector.class`，而 `String` 数组参数容纳 “`java.util.AbstractList`” 和 “`java.util.AbstractCollection`” 这两个 `String` 类型的数据。这样挺方便吧？

为什么只选择其一？

我已经探讨了对我而言，TestNG 的一些独有优势，但是它还有其他几个特性是 JUnit 所不具备的。例如 TestNG 中使用了测试分组，它可以根据诸如运行时间这样的特征来对测试分类。也可在 Java 1.4 中通过 javadoc 风格的注释来使用它，如 清单 5 所示。

正如我在本文开头所说，JUnit 4 和 TestNG 在表面上是相似的。然而，设计 JUnit 的目的是为了分析代码单元，而 TestNG 的预期用途则针对高级测试。对于大型测试套件，我们不希望在某一项测试失败时就得重新运行数千项测试，TestNG 的灵活性在这里尤为有用。这两个框架都有自己的优势，您可以随意同时使用它们。

追求代码质量: 驯服复杂的冗长代码

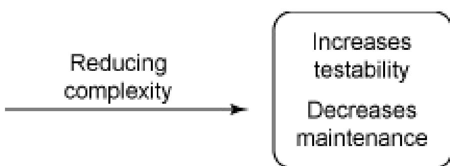
测量代码是否冗长的工具和度量

只是从远处看一眼乱七八糟四处蔓延的代码块，开发人员就会感到心惊肉跳 —— 这很正常！冗长的代码常常是复杂性的标志，会导致代码难以测试和维护。本月将学习三种测试代码复杂性的重要方法，它们分别基于方法长度、类长度和内部类耦合。在这一期的 追求代码质量 系列文章中，专家 Andrew Glover 将向您展示如何使用诸如 PMD 和 JavaNCSS 之类的工具，在您需要的时候获得更高的精度。

我毫不惭愧地承认，在看到复杂的代码块时，我也会感到恐惧和心里发毛。事实上，我敢说您在遇到大量方法和乱七八糟四处蔓延的类时，也会 有些心里发毛的。不能说在这些情况下寻求退路的人不是完人，这只是优秀开发人员的一种本能。过于复杂的代码难以测试和维护，这通常还意味着更高的出错率。

我在 本系列前面的文章 中已经解释了圈复杂性，它是令人讨厌的代码的一种先兆。具有高圈复杂度值的测试方法几乎总是把事情弄得一团糟，无法轻易收场。上一个月，我向您展示了如何使用 Extract Method 模式重构您的代码，从而将您带出迷宫。降低方法的复杂度可以使代码更易于测试和维护，如图 1 所示：

图 1. 降低复杂度可以使代码更易于维护和测试



不过，圈复杂性并不是确定高风险代码的惟一复杂性度量。您还可以利用类长度、方法长度和内部类耦合。这些度量之间存在着错综复杂的关联，但是很容易发现这些关联。这个月，我将解释它们为什么那么重要，以及如何使用 PMD 和 JavaNCSS 跟踪它们。

代码太多了！

了解简单代码和流畅代码之间的区别非常重要。简单代码不必过分简单或易于编写，只需易于理解 即可。您可以使用 C++ 编写简单代码，就像使用 Visual Basic 编写它们那样。不过，用任何语言反简化 代码的最快方式都是一次编写大量的代码。

提高代码质量

不要错过 Andrew 的 讨论论坛，可从中获得大多数紧急问题的答案。

考虑一下如何将此规则应用于方法和类。大多数人对记住信用卡号感到头疼的一个简单原因是，我们一次只能管理 7(±2) 片数据。了解了这一点，就会明白过多的条件会给以后带来麻烦，使测试和维护变得很困难。相同的原理也可以应用于逻辑块。

所有给定代码主体通常包含已分组的语句，它们拥有共同的目标，比如创建一个集合，将数据项添加到该集合中。但是，在一个长方法（long method）中分组数量众多的逻辑块可能会让人很快忘记该方法的总体意图，因为很少有人可以有效处理这样一个大的数据集。恰恰是这个缺点带来了代码基中的维护问题。冗长的方法是缺陷的避风港，因为很少有人可以有效地分析它们。长方法不仅完成太多的工作，也需要人们费很大的劲去理解！

就像长方法会让开发人员讨厌一样，长类（long class）也会令开发人员讨厌。相同的讨论也可以应用于总体代码，冗长的类可能会做太多的工作，并承担太多的责任。

什么样才算太长？

当然了，长方法或类的划分有点主观。有一个很有帮助的经验法则，您可以说非代码注释行超过 100 行的方法是长方法。不过，实际的数值是根据谈论的人而变化的。就我而言，截止点（cutoff point）大约是 50 行代码，但有些开发人员会说，如果某一方法需要您向下滚动整整一天才能看完，那么该方法太长了。截止点的定义取决于您自己。

类似地，您必须有自己的确定正确类大小的良好判断。许多人所提倡的一条经验法则是，类的代码行超过 1,000 行就可以说该类太长了。而另一些人则认为最好不要超过 500 行代码。

内部类耦合

对于一对象与其他对象之间的关系，复杂模式会不断重复其自身。对于导入许多外部依赖项或者拥有许多 `public` 方法的类，不但理解起来有些困难，而且所带来的责任重担的增加也会导致某种脆弱。

我将从依赖项开始。如果某一对象导入的外部类超过 80 个（不包括普通的 Java™ 系统库），那么就可以说该类具有高度输出耦合，这意味着更改导入的类可能会影响该类本身。在最糟糕的情况下，如果导入的是具体的类，并且它们的行为发生更改，那么执行导入的类可能会中断！（请参阅 参考资料，了解关于输出耦合的更多信息。）

观察对象导入的数量就很容易预测脆弱性，但如果使用 `.*` 符号（例如 `com.acme.user.*`）导入整个包，则很可能产生误导。为了更精确起见，可能需要注意对象所拥有的惟一类型 的数量（该数量可通过解析代码获得 —— 不是 `import` 语句）。如果应用程序的包结构大致上以某种在少数包中包含许多类的方式设计，则惟一类型度量（types metric）可能很有帮助。

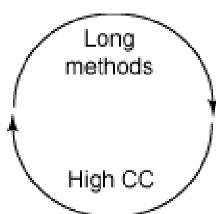
包含许多 `public` 方法的类也有许多导入。这些类通常会成为代码基的中心，就像 Facades 或工具类那样。因为存在这种责任（通过大量 `public` 方法导出），所以它们具有高度的输入耦合，也会导致反向的脆弱性。如果这些类中的任何一个发生更改，各种表面上不相关的

应用程序部分可能 发生中断。

复杂性是如何产生关联的

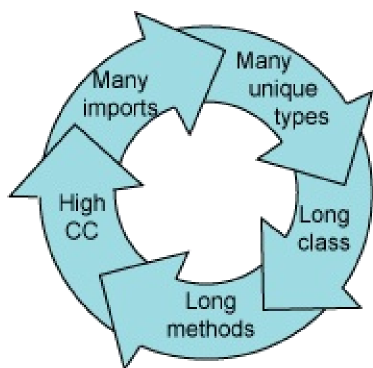
到目前为止，所给出的模式都在暗示臃肿的代码（长方法、太多的 public 方法、过多的条件和导入，等等）将影响可读性、可测试性和可维护性。因为该模式用各种度量来重复自己，所以所有这些因素都会导致相互关联。例如，长方法通常得容忍高圈复杂度值，如图 2 所示：

图 **2.** 长方法与圈复杂性相互关联



不过，相关性并不止于此。具有过多导入的类会有许多惟一类型。这些类通常非常大。而大型的类通常拥有长方法，长方法又常常有很高的圈复杂度值。图 3 展示了复杂性度量是如何相关的：

图 **3.** 复杂性度量是如何相关的



PMD 和 JavaNCSS

少量的繁琐代码可用 PMD 和（更小范围的）JavaNCSS 轻松处理，很容易结合使用这两种工具，以构建诸如 Ant 和 Maven 之类的平台。

可以将 PMD 看作是基于规则的引擎，它分析源代码并报告正被违反的某一规则的所有实例。PMD 目前定义了大约 200 个规则，其中一些特定规则是针对方法长度、类长度和惟一类型的，还有一些用于计算 `public` 方法。您还可以定义定制规则和修改现有规则（例如，为了反映域的需求）。

定制 PMD

例如，我将使用 PMD 的经过恰当命名的 ExcessiveMethodLength 规则来发现长方法。此规则的默认长度阈值是 100（这意味着如果某个所扫描方法的长度超过 100 行，则 PMD 会报告出现一个违规），但是如果您喜欢的话，可以降低该阈值。

PMD 规则可以定义属性，通过站在 PMD 开发团队的角度很好地进行预见，您可以通过使用规则集文件在运行的时候覆盖这些属性。要将 ExcessiveMethodLength 规则的默认值从 100 降低到 50，可以将 `properties` 元素添加到 `rule` 定义中并引用属性的名称。在清单 1 中，我将一个名为 `minimum` 的属性添加到了 PMD `rule` 定义中：

清单 **1.** 定制 **ExcessiveMethodLength** 规则

```
<rule ref="rulesets/codesize.xml/ExcessiveMethodLength">
 <properties>
  <property name="minimum" value="50"/>
 </properties>
</rule>
```

用 Ant 工具调用带有定制规则集文件的 PMD 需要通过 PMD 任务的 `rulesetfiles` 属性提供一条到该定制文件的路径，如清单 2 中所示：

清单 **2.** 引用定制规则集文件

```
<pmd rulesetfiles="./tools/pmd/rules-pmd.xml">
 <formatter type="xml" toFile="${defaulttargetdir}/pmd_report.xml"/>
 <formatter type="html" toFile="${defaulttargetdir}/pmd_report.html"/>
 <fileset dir="./src/java">
  <include name="**/*.java"/>
 </fileset>
</pmd>
```

PMD 报告由源文件导致的违规，正如您在图 4 中可以看到的，在本例中，只有少数几个方法的源代码行超过了 50 行：

图 **4. PMD Ant** 报告的示例

对于长类，PMD 有 ExcessiveClassLength 规则，长类的默认值为 1,000 行代码。对于 ExcessiveMethodLength 规则，很容易使用更适合的值覆盖默认值。此外，PMD 还有一个用来计算惟一类型的规则，即 CouplingBetweenObjects 规则。要计算导入，请参见 ExcessiveImports 规则。这两个规则都是可配置的。

使用 **JavaNCSS** 测量代码是否冗长

PMD 定义了用来分析源代码的特定规则，与 PMD 相对，JavaNCSS 分析代码基并报告所有一切 与代码长度相关的事项，包括类大小、方法大小和类中找到的方法数量。对于 JavaNCSS，阈值无关紧要，它计算所找到的每个文件并报告值，而不管 大小如何。尽管与 PMD 相比较而言，这类数据看起来似乎有些呆板（并且可能有点罗嗦！），但它有它存在的道理。

通过报告所有文件大小，JavaNCSS 使理解相关值成为可能，而 PMD 常常难以做到这一点。例如，PMD 只报告违规的文件，这意味着只理解部分代码基的数据，而 JavaNCSS 在上下文中提供了代码长度数据，如图 5 所示：

图 **5. JavaNCSS Ant** 报告的示例

JavaNCSS Analysis - Mozilla Firefox

File Edit View Go Bookmarks Tools Help

Packages

Nr.	Classes	Functions	NCSS	Javadocs	Package
1	8	38	223	40	com.acme.phnix.ant
2	2	11	75	10	com.acme.phnix.frmwrk.dependency
3	1	4	10	5	com.acme.phnix.frmwrk.dependency.exception
4	1	2	4	1	com.acme.phnix.frmwrk.filter
5	3	13	57	12	com.acme.phnix.frmwrk.filter.impl
6	1	0	3	1	com.acme.phnix.frmwrk.report
7	3	8	88	14	com.acme.phnix.frmwrk.report.xml
	19	76	460	83	Total

Packages	Classes	Functions	NCSS	Javadocs	|per
7.00	19.00	76.00	460.00	83.00	| Project
	2.71	10.86	65.71	11.86	| Package
		4.00	24.21	4.37	| Class
			6.05	1.09	| Function

Objects

Nr.	NCSS	Functions	Classes	Javadocs	Class
1	6	2	0	1	com.acme.phnix.ant.Dependencies
2	139	20	0	17	com.acme.phnix.ant.DependencyFinderTask
3	8	3	0	4	com.acme.phnix.ant.DependencyPath
4	10	4	0	5	com.acme.phnix.ant.RegexFilter
5	13	5	0	6	com.acme.phnix.ant.Report
6	10	4	0	5	com.acme.phnix.ant.SimpleFilter
7	1	0	0	1	com.acme.phnix.ant.SourceDependencies
8	1	0	0	1	com.acme.phnix.ant.TargetDependencies
9	19	6	0	5	com.acme.phnix.frmwrk.dependency.Dependency
10	43	5	0	5	com.acme.phnix.frmwrk.dependency.DependencyFinder
11	9	4	0	5	com.acme.phnix.frmwrk.dependency.exception.DependencyFinderException
12	3	2	0	1	com.acme.phnix.frmwrk.filter.Filter
13	8	3	0	2	com.acme.phnix.frmwrk.filter.impl.ClassInclusionFilter
14	22	6	0	7	com.acme.phnix.frmwrk.filter.impl.RegexPackageFilter
15	11	4	0	3	com.acme.phnix.frmwrk.filter.impl.SimplePackageFilter
16	1	0	0	1	com.acme.phnix.frmwrk.report.IDependencyReport
17	39	3	1	7	com.acme.phnix.frmwrk.report.xml.BatchDependencyXMLReport
18	13	2	0	3	com.acme.phnix.frmwrk.report.xml.DependencyXMLReport
19	16	3	0	4	com.acme.phnix.frmwrk.report.xml.XMLReportHelper
Average Object NCSS:					19.58
Average Object Functions:					4.00
Average Object Inner Classes:					0.05
Average Object Javadoc Comments:					4.37
Program NCSS:					460.00

Done

结束语

绿地开发（greenfield development）是指开发团队首先开发一个空白的 IDE 控制台，并用漂亮、简洁的代码填充它，这只是软件应用程序生存期中一个非常小 的片段。如今，很多跨国企业仍然在运行基于 COBOL 的应用程序，从开发人员的角度看，这意味着要与您不认识的人在很久以前编写的代码作斗争。

在遇到这样的难题时，通常会令人感到非常厌恶，您只能在连续几天的时间里声称自己生病了进行逃避。随后的某一时刻，您必须面对大量代码块并将它们搞定。使用针对类长度、方法长度和内部类耦合的复杂性度量（即对象导入和惟一类型）是理解您所面临的困难的第一步。从一些与类大小和方法大小有关的经验法则开始，然后使用诸如 PMD 和 JavaNCSS 之类的工具详细介绍。

当第一次在遗留代码基上使用复杂性度量时，您将了解到一个庞大的数量，但不要就此停住脚步。通过继续监视复杂性度量，您可以作出更明智的决定，并在不断扩展和维护代码时降低风险。

追求代码质量: 用代码度量进行重构

用代码度量和提取方法模式进行目的明确的重构

在 追求代码质量 的前一期中，学习了如何用代码度量客观地测量代码质量。这个月，Andrew Glover 将介绍如何使用相同的度量方法和提取方法模式进行有针对性的重构。

在我上中学的时候，有一位英语教师说：“写作就是重写别人已经 重写过的东西。” 直到大学，我才眞正理解了他这句话的意思。而且，当我自觉地采用这个实践的时候，就开始喜欢上了写作。我开始为我写的东西自豪。我开始眞正在意我的表达方式和要传达的内容。

当我开始开发人员生涯时，我喜欢阅读有经验的专家编写的技术书籍，而且想知道为什么他们花这么多时间编写代码。那时，编写代码看起来是件容易的工作 —— 有些人（总是比我级别高的人）会给我一个问题，而我会用任何可行的方法解决它。

直到我开始与其他开发人员合作大型项目，才开始理解我的技能的眞正意义所在。我也就在这个时候起，开始有意识地关心我编写的代码，甚至关心起其他人 编写的代码。现在我知道了，如果不注意代码质量，那么迟早它们会给我造成一团乱麻。

提高代码质量

不要错过 Andrew 的附带 讨论组 ，可以在这里得到最迫切问题的答案。

我恍然大悟 的一刻出现在 1999 年底，那时我正在阅读 Martin Fowler 那本影响重大的书 *Refactoring: Improving the Design of Existing Code*（重构：改进现有代码的设计，这本书对一系列重构模式进行分类，并由此建立了重构的公共词汇。在此之前，我一直都在重构我的代码（或者其他人的代码），但是却不知道自己做的就是重构。现在，我开始为我编写和重构的代码感到更加自豪，因为我做的工作正是在促进代码的编写方式并让它们日后更易维护。

什么是重构？

按照我的观点，重构就是改进已经改进的 代码的行为。实际上，重构是个永不停止的代码编写过程，它的目的是通过结构的改进而提高代码体的可维护性，但却不 改变代码的整体行为。重要的是要记住重构与重写 代码明显不同。

重写代码会修改代码的行为甚至合约，而重构保持对外接口不变。对于重构方法的客户机来说，看不到区别。事情像以前一样工作，但是工作得更好，主要是因为增强的可测试性或者明显的性能提升。

主动和被动重构

那么问题就变成了“我怎么才能知道什么时候该进行重构呢？”一段代码的可维护性是个主观的问题。但是，我们中的多数人都会发现，维护自己编写的代码要比维护其他人编写的代码容易得多。但在这点上也有争议 —— 在整个职业生涯中维护自己的代码是最大挑战。没有几个真正的“代码牛仔”足够幸运地能够不断地变换工作，而不必修改其他人的代码。对于我们中的多数人来说，必须维护其他人的代码恰恰是程序员生活的一部分。决定代码是否需要重构的方法，通常是主观的。

但是，也有可能客观地判断代码是否应当重构，不论是自己的代码还是别人的代码。在 这个系列前面的文章中，我介绍了如何用代码度量客观地测试代码质量。实际上，可以用代码度量很容易地找出可能难以维护的代码。一旦客观地判断出代码中有问题，那么就可以用方便的重构模式改进它。

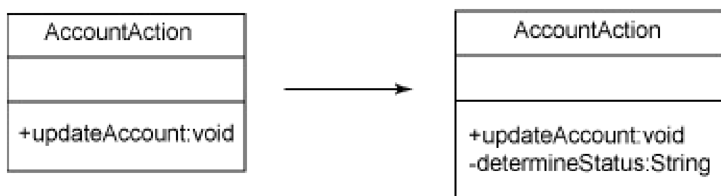
总是运行测试用例！

重构别人编写的代码的秘诀是不要把它弄得更糟。在我重构生涯的早期，学到的一件事就是在修改一些东西之前 拥有一个测试用例很重要。我是通过艰苦的一夜，在我自己整理得很好的重构方法中苦苦寻觅，只为找到一个我不小心破坏的别人编写的工作正常的代码之后学到这个教训的，不小心破坏的原因就在于重构之前没有对应的测试用例。请注意我的警告，在自己进行重构之前，总是要运行测试用例！

提取方法模式

Martin Fowler 的书出版之后的几年中，增加了许多新的重构模式分类；但是，迄今为止最容易学习的模式，也可能是最有效的模式，仍然是提取方法（*Extract Method*） 模式。在这个模式中，方法的一个逻辑部分被移除，并被赋予自己的方法定义。现在被移走的方法体被新方法的调用代替，如图 1 的 UML 图所示：

图 **1.** 提取方法模式实践



提取方法模式提供了两个关键好处：

- 原来的方法现在更短了，因此也更容易理解。
- 移走并放在自己方法中的逻辑体现在更容易测试。

降低圈复杂度

在使用的时候，对于被高度圈复杂度值感染的方法来说，提取方法是一剂良药。您可能会记得，圈复杂度通过度量方法的路径数量；所以，可以认为如果提取 出其中一些路径，重构方法的整体复杂性会降低。

例如，假设在运行了像 PMD 这样的代码分析工具之后，结果报告显示其中一个类包含的一个方法有较高的圈复杂度值，如图 2 所示：

图 **2.** 圈复杂度值高达 **23**！

Avoid really long methods.	285
Avoid really long parameter lists.	285
The method 'updateContent' has a Cyclomatic Complexity of 23.	285
Avoid reassigning parameters such as 'title'	289
Avoid reassigning parameters such as 'url'	291

在仔细查看了这个方法之后，发现这个方法过长的原因是使用了太多的条件逻辑。正如我以前在这个系列中指出的（请参阅 参考资料），这会增加方法中产生缺陷的风险。谢天谢地，`updateContent()` 方法还有个测试用例。即使已经认为这个方法有风险，测试也会减轻一些 风险。

另一方面，测试已经精心地编写成可以测试 `updateContent()` 方法中的 23 个路径。实际上，好的规则应当是：应当编写至少 *23* 个测试。而且，要想编写一个测试用例，恰好能隔离出方法中的第 18 个条件，那将是极大的挑战！

小就是美

是否真的要测试长方法中的第 18 个条件，是个判断问题。但是，如果逻辑中包含真实的业务值，就会想到测试它，这个时候就可以看到提取方法模式的作用了。要把风险降到最小很简单，只需把条件逻辑分解成更小的片段，然后创建容易测试的新方法。

例如，`updateContent()` 方法中下面的这小段条件逻辑创建一个状态 `String`。如清单 1 所示，逻辑的隔离看起来足够简单：

清单 **1.** 条件逻辑成熟到可以进行提取

```
//...other code above

String retstatus = null;
if ( lastChangedStatus != null && lastChangedStatus.size() > 0 ){
 if ( status.getId() == ((IStatus)lastChangedStatus.get(0)).getId() ){
  retstatus = "Change in Current status";
 }else{
  retstatus = "Account Previously Changed in: " +
    ((IStatus)lastChangedStatus.get(0)).getStatusIdentification();
 }
}else{
  retstatus = "No Changes Since Creation";
}

//...more code below
```

通过把这一小段条件逻辑提取到简洁的新方法中（如清单 2 所示），就做到了两件事：一，把 `updateContent()` 方法的整体复杂性降低了 5；二，逻辑的隔离很完整，可以容易地对它进行测试。

清单 **2.** 提取方法产生 **getStatus**

```
private String getStatus(IStatus status, List lastChangedStatus) {
  String retstatus = null;
  if ( lastChangedStatus != null && lastChangedStatus.size() > 0 ){
    if ( status.getId() == ((IStatus)lastChangedStatus.get(0)).getId() ){
      retstatus = "Change in Current status";
    }else{
      retstatus = "Account Previously Changed in: " +
        ((IStatus)lastChangedStatus.get(0)).getStatusIdentification();
    }
  }else{
    retstatus = "No Changes Since Creation";
  }
  return retstatus;
}
```

现在可以把 `updateContent()` 方法体中的一部分替换成对新创建的 `getStatus()` 方法的调用，如清单 3 所示：

清单 **3.** 调用 **getStatus**

```
//...other code above

String iStatus = getStatus(status, lastChangedStatus);

//...more code below
```

请记住运行现有的测试，以验证什么都没被破坏！

测试私有方法

您将注意到在 清单 2 中定义的新 `getStatus()` 方法被声明为 `private`。这在想验证隔离的方法的行为的时候就形成了一个有趣的挑战。有许多方法可以解决这个问题：

- 把方法声明成 `public`。
- 把方法声明成 `protected`，并把测试用例放在同一个包中。
- 在父类中建立一个内部类，这个内部类是个测试用例。

还有另一个选择：保留方法现有的声明不变（即 `private`），并采用优秀的 JUnit 插件项目来测试它。

PrivateAccessor 类

JUnit 插件项目有一些方便的工具，可以帮助 JUnit 进行测试。其中最有用的一个就是 `PrivateAccessor` 类，它把对 `private` 方法的测试变成小菜一碟，无论选择的测试框架是什么。`PrivateAccessor` 类对 JUnit 没有显式的依赖，所以可以把它用于任何测试框架，例如 TestNG。

`PrivateAccessor` 的 API 很简单 —— 向 `invoke()` 方法提供方法的名称（作为 `String`）和方法对应的参数类型和相关的值（分别在 `Class` 和 `Object` 数组中），就会返回被调用方法的值。在幕后，`PrivateAccessor` 类实际上利用 Java 的反射 API 关闭了对象的可访问性。但是请记住，如果虚拟机有定制的安全性设置，那么这个工具可能无法正确工作。

在清单 4 中，调用 `getStatus()` 方法时两个参数值都设置为 `null`。`invoke()` 方法返回一个 `Object`，所以要转换成 `String`。还请注意 `invoke()` 方法声明它要 `throws Throwable`，必须捕获异常或者让测试框架处理它，就像我做的那样。

清单 4. 测试私有方法

```
public void testGetStatus() throws Throwable{
  AccountAction action = new AccountAction();

  String value = (String)PrivateAccessor.invoke(action,
      "getStatus", new Class[]{IStatus.class, List.class},
       new Object[]{null, null});

  assertEquals("should be No Changes Since Creation",
    "No Changes Since Creation", value);
}
```

请注意 `invoke()` 方法被覆盖成可以接受一个 `Object` 实例（如清单 4 所示）或一个 `Class`（这时期望的 `private` 方法也是 `static` 的）。

还请记住，使用反射调用 `private` 方法会对生成的结果带来一定程度的脆弱性。如果有人改变了 `getStatus()` 方法的名字，以上测试就会失败；但是，如果经常测试，就可以迅速地进行适当的修正。

结束语

在抗击圈复杂度时，请记住大部分编写到应用程序中的路径是应用程序的整体行为所固有的。也就是说，很难显著地减少路径的整体数量。重构只是把这些路径放在更小的代码段中，从而更容易测试。这些小的代码段也更容易维护。

追求代码质量: 软件架构的代码质量

使用耦合度量来支持系统架构

大多数设计良好的软件架构都趋向于支持系统的可扩展性、可维护性和可靠性。遗憾的是，对质量问题的疏忽极可能使软件架构师的努力白费。在追求代码质量 系列的这一期文章中，质量专家 Andrew Glover 解释如何持续地监视并纠正会影响软件架构的长期生存能力的代码质量方面。

上一期文章中，我展示了如何使用代码度量来评估代码质量。尽管在那一期介绍的圈复杂度针对低级细节，如方法中执行路径的数量，但其他类型的度量针对的是代码的更高级方面。在本期文章中，我将展示如何使用各种耦合度量 来分析和支持软件架构。

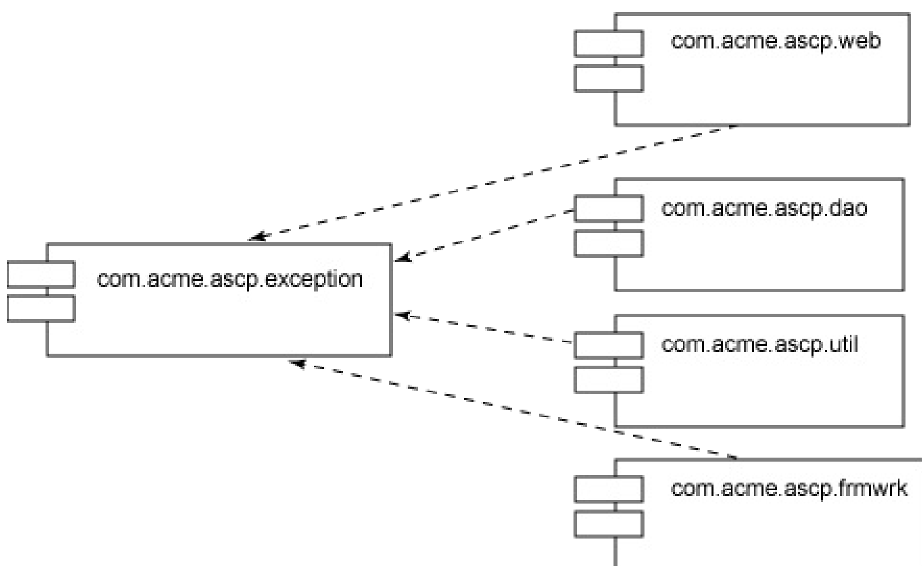
我将从两个比较有趣的耦合度量开始，即传入耦合 和传出耦合。这些基于整数的度量表示几个相关对象（即相互协调以产生行为的对象）。任一度量中高数值表示架构的维护问题：高传入耦合表示对象具有太多职责，而高传出耦合表示对象不够独立。在本期文章中，我将介绍每个这样的问题及其解决的方法。

传入耦合

具有太多职责并非什么坏事。例如，组件（或包）通常试图用于整个架构中，这就会给它们带来高传入耦合值。核心框架（如 Strut）、登录包（如 log4j）之类的实用工具以及异常层次结构通常具有高传入耦合。

在图 1 中，可以看到一个包 `com.acme.ascp.exception` 具有一个值为 4 的传入耦合。这并不奇怪，因为 `web` 、 `dao` 、 `util` 和 `frmwrk` 包都希望利用一个公共的异常框架。

图 **1.** 传入耦合的符号



如图 1 所示，`exception` 包具有一个值为 4 的传入耦合（或者叫做 *Ca*），这并非是件坏事。异常层次结构很少会出现很大的改变。监视 `exception` 包的传入耦合是个好主意，然而，由于彻底改变了这个包中的行为或契约，所以将引起它的四个依赖包全都出现连锁反应。

测量抽象性

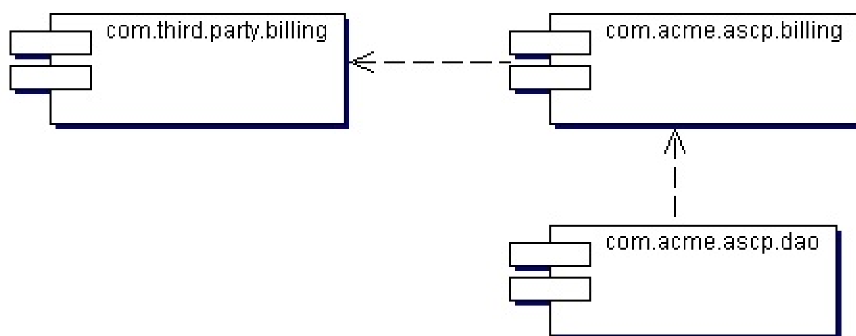
通过进一步检查 `exception` 包并注意抽象到具体类的比率，可以派生出另一个度量：抽象性。在本例中，`exception` 包具有零抽象性，因为它的所有类都是具体的。这与我前面的观察是一致的：`exception` 包中的高度具体性表示对 `exception` 作出的任何更改将影响所有相关包，即 `com.acme.ascp.frmwrk`、`com.acme.ascp.util`、`com.acme.ascp.dao` 和 `com.acme.ascp.web`。

通过理解传入耦合表示组件的职责，并持续监视这个度量，可以防止软件架构出现熵（*entropy*），即使在大多数设计良好的系统中也很容易出现熵。

支持设计灵活性

很多架构设计在利用第三方包时都考虑到了灵活性。获得灵活性最好是通过使用接口来防止架构在第三方包中发生更改。例如，系统设计师可以创建一个内部接口包 来利用第三方记帐代码，但是只对这些使用记帐代码的包公开接口。顺便说一下，这与 JDBC 的工作原理类似。

图 **2.** 通过设计获得灵活性



如图 2 所示，acme.ascp 应用程序通过 `com.acme.ascp.billing` 包与第三方记帐包相耦合。这创建了一定级别的灵活性：如果有了另一个第三方记帐包更加有利用价值，那么应该只有一个包会受到变更的影响。此外，`com.acme.ascp.billing` 的抽象性值是 0.8，这表明它可以通过接口和抽象类来防止被修改。

如果要转换到第三方实现，只需要对 `com.acme.ascp.billing` 包进行重构。更好的方法是：通过在设计中考虑灵活性以及了解变更的隐含意义，可以通过开发人员测试来防止修改所造成的任何损害。

在对内部记帐包作出变更前，您可以分析代码覆盖率报告，以确定是否有任何测试眞正 测试了这个包。找到一些级别的覆盖率后，您可以更仔细地检查这些测试案例来验证它们是否足够了。如果未找到覆盖率，您将会知道，关闭并插入新库的努力将更具风险性并可能花费更长的时间。

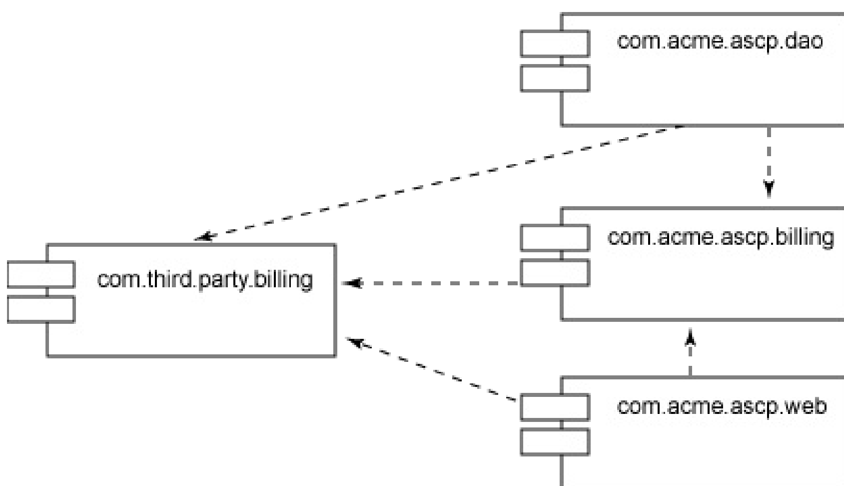
使用代码度量收集所有这些似乎正确的信息非常容易。另一方面，如果您根本不了解与测试覆盖率相关的包耦合的知识，那么为替换第三方库确定的时间最多就是个猜测！

监视熵

前面提到过，即使计划得最好的架构也会出现熵。通过团队磨损或未充分记录的意图，没有经验的开发人员可能会疏忽地导入似乎有用的包，不久以后，系统传入耦合的值将开始增长。

例如，将图 3 与 图 2 进行比较。您注意到架构增加的脆弱性了吗？现在不仅 `dao` 包直接利用第三方记帐包，而且另一个甚至不想直接使用任何记帐代码的包也引用了这两个记帐包！

图 **3.** 出现代码熵



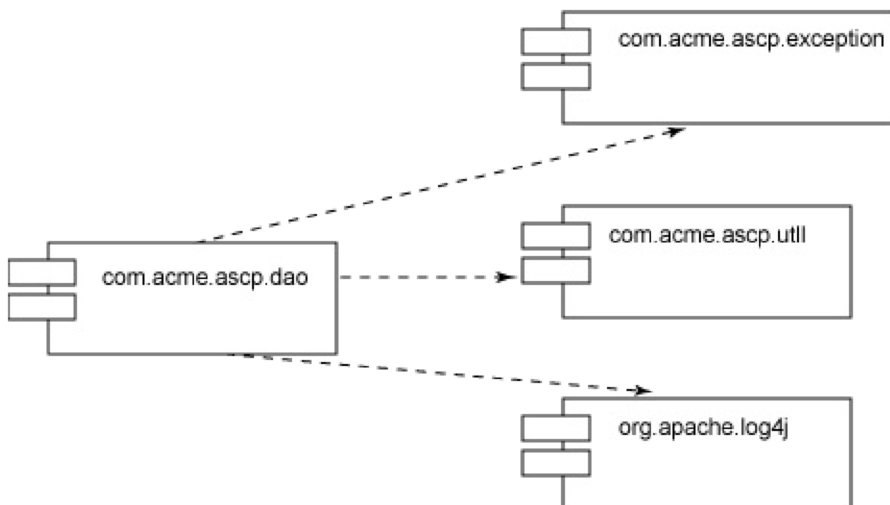
试图为另一个包关闭 `com.third.party.billing` 包确实很具有挑战性！设想一下降低产生缺陷和中断系统各种行为方面的风险所需的测试脚手架。事实上，像这样的架构很少变更，因为它们无法支持修改。更糟糕的是，即使像对现有组件的升级这样的重要修改也会导致整个代码基中断的事情出现。

传出耦合

如果传入耦合是一些依赖于某个特定组件的组件的话，那么传出耦合则是某个特定组件所依赖的一些组件。可以把传出耦合看作传入耦合的逆转。

对于更改如何影响代码来说，传出耦合的引号意义与传入耦合的类似。例如，图 4 描述了 `com.acme.ascp.dao` 包，它具有一个值为 3 的传出耦合（或者叫做 *Ce*）：

图 **4. dao** 包中的传出耦合



如图 4 所示，`com.acme.ascp.dao` 包依赖于 `org.apache.log4j` 、`com.acme.ascp.util` 和 `com.acme.ascp.exception` 组件来履行其行为契约。与传入耦合中一样，依赖性级别本身并不是什么坏事。重要的是您对耦合的了解以及耦合如何影响对相关组件的更改。

与传入耦合一样，抽象性度量在传出耦合中起作用。在 图 4 中，`com.acme.ascp.dao` 包完全是具体的；因此它的抽象性为 0。这表示其传出耦合包含 `com.acme.ascp.dao` 的组件自己会变得脆弱，因为 `com.acme.ascp.dao` 包与 3 个附加的包具有传出耦合。如果它们中的一个（比如说 `com.acme.ascp.util`）发生更改，将会在 `com.acme.ascp.dao` 中发生连锁反应。因为 `dao` 无法通过接口或抽象类隐藏注入细节，所以任何更改都可能影响它的依赖组件。

耦合与覆盖相加等于……

检查传出耦合的关系数据，并将其与代码覆盖相关联，会促进作出更明智的决策。例如，假设一个新的需求传达给开发团队。您可以将与该需求相关的更改精确到图 4 所示的 `com.acme.ascp.util` 包。而且，在以前几个版本中，依赖于 `util` 并且具有 0 抽象性的 `dao` 包具有很多高优先权的缺陷（非常可能是因为对这个包进行的开发人员测试太有限，有意思的是，最大的可能是由于代码中的高复杂性值引起的）。

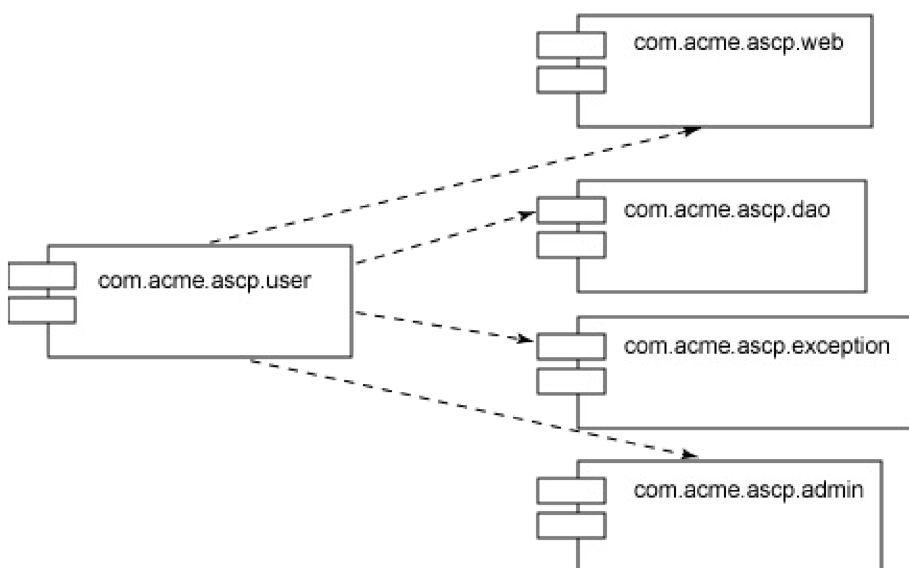
在这种情况下，您的优势是了解 `com.acme.ascp.util` 和 `com.acme.ascp.dao` 之间的关系。知道 `dao` 包依赖于 `util` 这一事实告诉您，为支持新需求而在 `util` 中进行的任何修改可能会对易出故障的 `dao` 包产生负面的影响！

看到这个链接将帮助进行风险评估，甚至帮助进行工作级别的分析。如果未注意到这个链接，您可能已经猜到将需要快速编码工作来支持新需求。如果已经看到这个链接，就可以分配适当的时间和资源来降低 `dao` 包中的间接损害。

监视依赖性

正像连续地监视传入耦合可以揭示架构设计中的熵一样，监视传出耦合也有助于发现不必要的依赖性。例如，在图 5 中，似乎在一些地方有人决定 `com.acme.ascp.web` 包要为 `com.acme.ascp.user` 提供内容。在 `user` 包中的某处，一个或多个对象正在实际从 `web` 包导入一个对象。

图 **5. user** 包中的传出耦合



很明显，这并非架构设计最初意图。但是，由于您一直针对传出耦合而监视系统，所以可以轻松地重构并改正这些不一致。或许，`web` 包中有用的实用工具对象应该移动到实用工具包，以便其他包可以利用它而不会引起不必要的依赖性。

测量不稳定性

您可以将系统的传出耦合和传入耦合的数量结合起来，形成另一个度量：不稳定性。通过将传出耦合除以传出传入耦合的和（ `Ce / (Ca + Ce)` ），可以生成一个比率，表示一个稳定的包（值接近于 0）或者不稳定的包（值接近于 1）。如这个等式所示，传出耦合对包的稳定性起作用：一个包越依赖于其他包，面对更改时它越容易受到连锁反应的影响。反过来说，一个包越被依赖，它越不可能发生更改。

例如，在图 5 中，`user` 包的不稳定性值为 1，这表示它有一个值为 4 的传出耦合，而没有传入耦合。像 `com.acme.ascp.dao` 这样的包中的更改将会 影响 `user` 包。

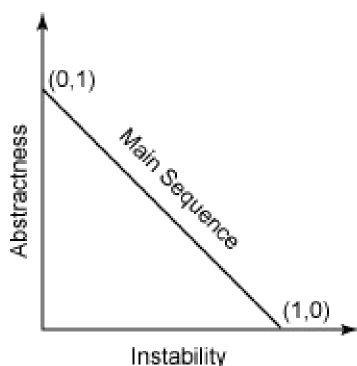
在设计和实现架构时，依赖于稳定的包是有益的，因为这些包不太可能更改。类似地，不稳定的包依赖性会在发生更改时增大架构内发生间接损害的风险。

到主序列的距离

到目前为止，我已经介绍了传入耦合和传出耦合，前者可用来评估更改包造成的影响，后者可用来评估外界的更改如何影响包。我还谈及了抽象性度量和不稳定性度量，前者在您想要了解如何轻松地对包进行修改时很有用，后者可用来了解包依赖性如何影响某个特定的包。

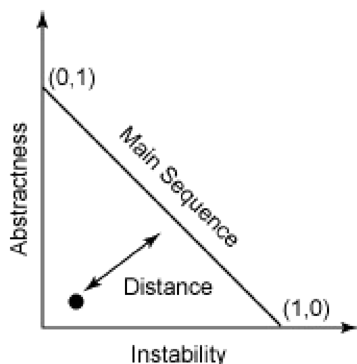
还可以使用另一个度量来了解影响软件架构的因素。这个度量通过 X, Y 坐标上的一条直线来平衡抽象性和不稳定性。主序列 是笛卡儿坐标上从 `X=0` 和 `Y=1` 到 `X=1` 和`Y=0` 的一条直线，如图 6 所示：

图 **6.** 主序列



通过沿着这条直线绘制包并测量它们到主序列的距离，可以推断包的平衡。如果包对于抽象性和不稳定性是平衡的，它的距离就接近于 0，如果包不平衡，那么它距离主序列的距离就接近于 1，如图 7 所示：

图 **7.** 到主序列的距离



检查到主序列的距离 度量会产生有趣的结果。例如，上面的 `user` 包生成的值为 0。就一个实现包来说，这个包是平衡的，即高度不稳定。

总体来说，“到主序列的距离” 度量尝试补偿实际实现。没有代码基包含所有抽象性和不稳定性值为 1 或 0 的包 —— 多数包的值位于这两个数字之间。通过监视 “到主序列的距离” 度量，可以衡量包是否正在变得不平衡。寻找偏远的值，如值接近于 1 的包（这表示它们距离主序列尽可能地远），有助于了解特定的不平衡如何影响架构的可维护性（例如，通过脆弱性）。

结束语

在本期文章中，您已经了解几种可以持续监视的架构度量。通过代码分析工具可以报告传入和传出耦合、不稳定性、抽象性、到主序列的距离，这些代码分析工具包括 JDepend、JarAnalyzer 和 Metrics plug-in for Eclipse（参见 参考资料）。监视系统的代码耦合度量有助于您掌握可破坏架构的常见趋势，即设计刚度、包熵和不必要的依赖性。此外，根据抽象性和不稳定性来测量系统平衡将使您不断的了解系统的可维护性。

让开发自动化: 除掉构建脚本中的气味

创建一致、可重复、可维护的构建

您把多少时间花在维护项目构建脚本上？也许远远超出您预期的或者可以忍受的时间。其实大可不必遭受如此痛苦的经历。在这一期的让开发自动化 中，开发自动化专家 Paul Duvall 将演示如何改进很多常见的妨碍团队创建一致的、可重复的、可维护的构建的实践。

当描述代码之类的东西时，我不喜欢 “气味（smell）”这个词。因为用拟人的手法来谈论比特和字节往往令人觉得很怪异。并不是说“气味”这个词不能准确地反映出某种表明代码可能有错误的症状，只是我觉得这样听起来很滑稽。然而，我依然选择再次用这种令人厌烦的方式来描述软件构建，坦白说，这是因为这些年我见过的很多构建脚本都散发着难闻的气味。

在创建构建脚本时，即使是伟大的程序员也常常会遇到困难。就好像最近才学会如何编写程序性 代码似的 —— 他们还会编写庞大的单块构建文件、通过复制-粘贴编写代码、对属性进行硬编码等等。我总是很想知道为什么会这样。也许是因为构建脚本没有被编译成客户最终会使用的东西？然而我们都知道，要创建客户最终使用的代码，构建脚本是中心，如果那些脚本败絮其中，那么要想有效地创建 软件，就需要克服重重挑战。

幸运的是，您可以轻松地在构建（不管是 Ant、Maven 还是定制的）之上部署一些实践，它们虽然可以帮助您创建一致的、可重复的、可维护的构建，但其过程会很长。学习如何创建更好的构建脚本的一种有效的方法是搞清楚哪些事情不要 去做，理解其中的道理，然后看看做事的正确 方法。在本文中，我将详细论述您应该避免的 9 种最常见的构建中的气味，为什么应该避免它们，以及如何修复它们：

- 惟 IDE 的构建
- 复制-粘贴式的编写脚本方法
- 冗长的目标
- 庞大的构建文件
- 没有清理干净
- 硬编码的值
- 测试失败还能构建成功
- 魔力机
- 格式的缺失

关于本系列

作为一名开发人员，我们的工作就是为用户将过程自动化。然而，我们当中有很多人却忽视了将我们自己的开发过程自动化的机会。为此，我编写了 让开发自动化 这个系列的文章，专门探索软件开发过程自动化的实际应用，并教您何时以及如何成功地应用自动化。

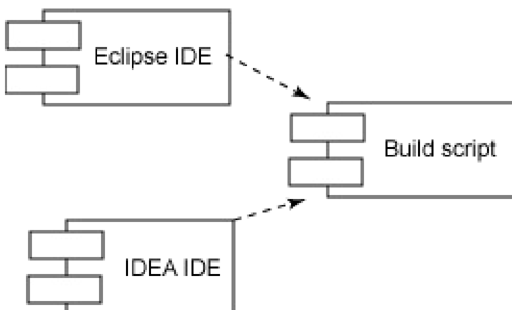
这里无意给出完整的列表，不过这份列表的确代表了近年来我读过的和写过的构建脚本中，我遇到的较为常见的一些气味。有些工具，例如 Maven，是为处理与构建有关的很多管道而设计的，它们可以帮助减轻部分气味。但是无论使用什么工具，还是有很多问题会发生。

避免惟 IDE 的构建

惟 IDE（IDE-only）的构建是指只能通过开发人员的 IDE 执行的构建，不幸的是，这似乎在构建中很常见。惟 IDE 的构建的问题是，它助长了“在我的计算机上能运行”问题，即软件在开发人员的环境中可以运行，但是在任何其他人的环境中就不能运行。而且，由于惟 IDE 构建自动化程度不是很高，因而为集成到持续集成（Continuous Integration）环境带来极大的挑战。实际上，没有人为的干预，惟 IDE 常常无法自动化。

我们要清楚：使用 IDE 来执行构建并没有错，但是 IDE 不应该成为能构建软件的惟一环境。特别是，一个完全用脚本编写的构建，可以使开发团队能够使用多种 IDE，因为只存在从 IDE 到构建的依赖性，而不存在相反方向的依赖性，如图 1 所示：

图 **1. IDE** 与构建的依赖关系



惟 IDE 的构建有碍自动化，清除的惟一方法就是创建可编写脚本的构建。有足够的文档和太多的书籍可以为您提供指导（见 参考资料），而像 Maven 之类的项目也为从头开始定义构建提供了极大的方便。不管采用何种方法，都是选择一种构建平台，然后尽快地让项目成为可编写脚本的。

复制-粘贴就像廉价的香水

复制代码是软件项目当中一个常见的问题。实际上，甚至很多流行的开放源码项目都存在 20% 到 30% 的复制代码。代码复制令软件程序更难于维护，同理，构建脚本中的复制代码也存在这样的问题。例如，想象一下，假设您需要通过 Ant 的 `fileset` 类型引用特定的文件，如清单 1 所示：

清单 **1.** 复制-粘贴 **Ant** 脚本

```
<fileset dir="./brewery/src" >
  <include name="**/*.java"/>
  <exclude name="**/*.groovy"/>
</fileset>
```

如果需要在其他地方引用这组文件，例如为了编译、检查或生成文档，那么最终您可能会在多个地方使用相同的 `fileset`。如果在将来某个时候，您需要对那个 `fileset` 做出修改（比如说排除 `.groovy` 文件），那么最终可能需要在多个地方做更改。显然，这不是可维护的解决方案。然而，要除掉这股气味其实很简单。

如清单 2 所示，通过 Ant 的 `patternset` 类型可以引用一个逻辑名称，以表示所需要的文件。那么，当需要向 `fileset` 添加（或排除）文件时，只需更改一次。

清单 **2.** 复制**-**粘贴 **Ant** 脚本

```
<patternset id="sources.pattern">
  <include name="**/*.java"/>
  <exclude name="**/*.groovy"/>
</patternset>
...
<fileset dir="./brewery/src">
  <patternset refid="sources.pattern"/>
</fileset>
```

对于精通面向对象编程的人来说，这种修复方法看上去很熟悉：既定的惯例不是在不同的类中一次又一次地定义相同的逻辑，而是将那个逻辑放在一个方法中，在不同地方都可以调用这个方法。于是，这个方法成为惟一的维护点，从而可以限制错误级联并可以鼓励重用。

不要掺入冗长目标的气味

Martin Fowler 在他撰写的 *Refactoring* 这本书中，对代码中存在冗长方法的气味这个问题做了精妙的描述 —— 过程越长，越难理解。实际上，冗长方法最终会担负太多的责任。当谈到构建时， 冗长目标这种构建气味是指更难于理解和维护的脚本。清单 3 就展示了一个相当冗长的目标：

清单 **3.** 冗长目标

```
<target name="run-tests">
  <mkdir dir="${classes.dir}"/>
  <javac destdir="${classes.dir}" debug="true">
    <src path="${src.dir}" />
    <classpath refid="project.class.path"/>
  </javac>
  <javac destdir="${classes.dir}" debug="true">
    <src path="${test.unit.dir}"/>
    <classpath refid="test.class.path"/>
  </javac>
  <mkdir dir="${logs.junit.dir}" />
  <junit fork="yes" haltonfailure="true" dir="${basedir}" printsummary="yes">
    <classpath refid="test.class.path" />
    <classpath refid="project.class.path"/>
    <formatter type="plain" usefile="true" />
    <formatter type="xml" usefile="true" />
    <batchtest fork="yes" todir="${logs.junit.dir}">
      <fileset dir="${test.unit.dir}">
        <patternset refid="test.sources.pattern"/>
      </fileset>
    </batchtest>
  </junit>
  <mkdir dir="${reports.junit.dir}" />
  <junitreport todir="${reports.junit.dir}">
    <fileset dir="${logs.junit.dir}">
      <include name="TEST-*.xml" />
      <include name="TEST-*.txt" />
    </fileset>
    <report format="frames" todir="${reports.junit.dir}" />
  </junitreport>
</target>
```

这个冗长的目标（相信我，我还见过冗长得多的目标）要执行四个不同的过程：编译源代码、编译测试、运行 JUnit 测试和创建一个 JUnitReport。要担负的责任已经够多了，更不用说将所有 XML 放在一个地方所增加的相关的复杂性。实际上，这个目标可以拆分成四个不同的、逻辑上的目标，如清单 4 所示：

清单 **4.** 提取目标

```
 <target name="compile-src">
    <mkdir dir="${classes.dir}"/>
    <javac destdir="${classes.dir}" debug="true">
      <src path="${src.dir}" />
      <classpath refid="project.class.path"/>
    </javac>
  </target>

  <target name="compile-tests">
    <mkdir dir="${classes.dir}"/>
    <javac destdir="${classes.dir}" debug="true">
      <src path="${test.unit.dir}"/>
      <classpath refid="test.class.path"/>
    </javac>
  </target>

  <target name="run-tests" depends="compile-src,compile-tests">
    <mkdir dir="${logs.junit.dir}" />
    <junit fork="yes" haltonfailure="true" dir="${basedir}" printsummary="yes">
      <classpath refid="test.class.path" />
      <classpath refid="project.class.path"/>
      <formatter type="plain" usefile="true" />
      <formatter type="xml" usefile="true" />
      <batchtest fork="yes" todir="${logs.junit.dir}">
        <fileset dir="${test.unit.dir}">
          <patternset refid="test.sources.pattern"/>
        </fileset>
      </batchtest>
    </junit>
  </target>

  <target name="run-test-report" depends="compile-src,compile-tests,run-tests">
      <mkdir dir="${reports.junit.dir}" />
      <junitreport todir="${reports.junit.dir}">
      <fileset dir="${logs.junit.dir}">
        <include name="TEST-*.xml" />
        <include name="TEST-*.txt" />
      </fileset>
      <report format="frames" todir="${reports.junit.dir}" />
    </junitreport>
  </target>
```

可以看到，由于每个目标只担负一种责任，清单 4 中的代码理解起来要容易得多。根据用途分离目标，不但可以减少复杂性，还为在不同上下文中使用目标创造了条件，必要时还可以重用。

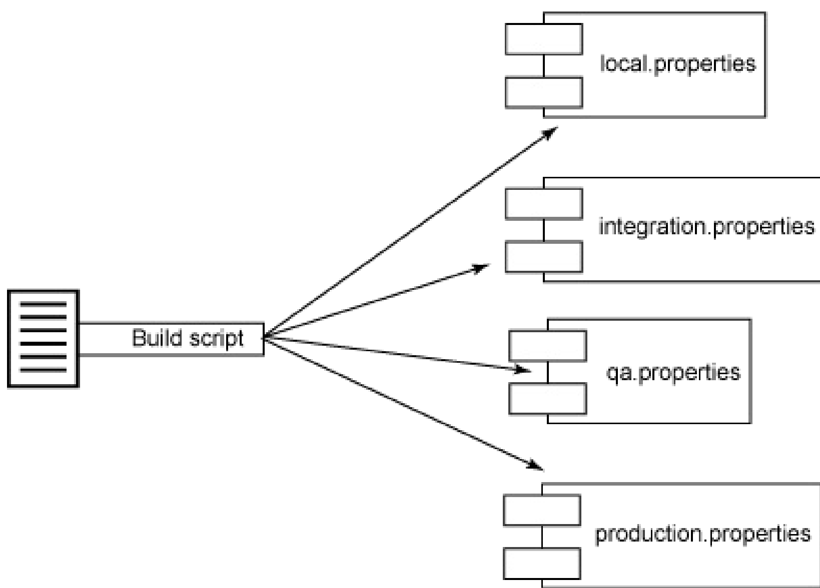
庞大的构建文件也有一种很重的气味

Fowler 还将 庞大的类也看作一种代码气味。就构建脚本而言，有这种类似气味的就是庞大的构建文件，它相当难以读懂。很难知道哪个目标是做什么的，目标的依赖关系是什么。这同样会给维护带来问题。而且，庞大的构建文件通常有相当多的剪切-粘贴的痕迹。

为了缩小构建文件，可以从脚本中找出逻辑上相关的部分，将它们提取到更小的构建文件中，由主构建文件来执行这些较小的构建文件（例如，在 Ant 中，可以使用 `ant` 任务调用其他构建文件）。

通常，我喜欢根据核心功能拆分构建脚本，确保它们可以作为独立脚本来执行（想想构建组件化）。例如，在我的 Ant 构建中，我喜欢定义四种类型的开发者测试：单元、组件、系统和功能。而且，我还喜欢运行四种类型的自动检查工具：编码标准、依赖性分析、代码覆盖范围和代码复杂度。我不是将这些测试和检查工具的执行放在一个庞大的构建脚本中（还加上编译、数据库集成和部署），而是将测试和检查工具的执行目标提取到两个不同的构建文件中，如图 2 所示：

图 **2.** 提取构建文件



更小、更简洁的构建文件维护和理解起来要容易得多。实际上，这种模式对于代码而言同样适用。我们似乎在这里看到了模式的概念，不是吗？

没有清理

没有严格减少所有底层假设的构建无疑是一颗定时炸弹。例如，如果构建没有避免一些简单的假设，例如会去掉用陈旧的数据生成的二进制文件，那么前一次构建遗留下来的文件就会引起错误。或者，正是由于前一次构建留下的文件，构建竟然得以"成功"，这种情况更糟糕。

幸运的是，这个问题的解决办法很直观：只需删除任何之前的构建留下的所有目录和文件，就可以很容易地消除假设。这个简单的动作就可以减少假设，保证构建的成功或失败都是正确的。清单 5 演示了通过使用 `delete` Ant 任务删除之前的构建所使用的所有文件或目录，从而清理构建环境的一个例子：

清单 **5.** 事先清理

```
<target name="clean">
  <delete dir="${logs.dir}" quiet="true" failonerror="false"/>
  <delete dir="${build.dir}" quiet="true" failonerror="false"/>
  <delete dir="${reports.dir}" quiet="true" failonerror="false"/>
  <delete file="cobertura.ser" quiet="true" failonerror="false"/>
</target>
```

众所周知，旧的构建遗留下来的文件会导致很多不必要的麻烦。为了自己的方便，在运行一个构建之前，务必先删除构建所创建的任何工件。

硬编码的臭味

复制-粘贴式的编程有碍重用，将值进行硬编码又何尝不是呢。当构建脚本包含硬编码的值时，如果某个方面需要修改，那么就需要在多个地方修改那个值。更糟糕的是，很可能会忽略了某个地方而没有改那个值，从而引起与不匹配的值相关的错误，这种错误是很隐蔽的。而且，如果相信我的建议，选择使用多个构建脚本，那么硬编码的值将可能会成为构建维护中最终的挑战。在这一点上也请相信我！

例如，在清单 6 中，`run-simian` 任务有很多硬编码的路径和值，即 `_reports` 目录：

清单 **6.** 硬编码的值

```
  <target name="run-simian">
    <taskdef resource="simiantask.properties"
      classpath="simian.classpath" classpathref="simian.classpath" />
    <delete dir="./_reports" quiet="true" />
    <mkdir dir="./_reports" />
    <simian threshold="2" language="java"
      ignoreCurlyBraces="true" ignoreIdentifierCase="true" ignoreStrings="true"
      ignoreStringCase="true" ignoreNumbers="true"  ignoreCharacters="true">
      <fileset dir="${src.dir}"/>
      <formatter type="xml" toFile="./_reports/simian-log.xml" />
    </simian>
    <xslt taskname="simian"
      in="./_reports/simian-log.xml"
      out="./_reports/Simian-Report.html"
      style="./_config/simian.xsl" />
  </target>
```

如果硬编码 `_reports` 目录，那么当我决定将 Simian 报告放到另一个目录时，就会很麻烦。而且，如果其他工具在脚本的其他地方使用这个目录，那么很可能会有人输错目录名称，导致报告显示在不同的目录中。这时可以定义一个属性值，由这个属性值指向这个目录。然后，在整个脚本中都可以引用这个属性，这意味着当需要更改的时候，只需光顾一个地方，即属性的定义。清单 7 展示了重构之后的 `run-simian` 任务：

清单 **7.** 使用属性

```
<target name="run-simian">
  <taskdef resource="simiantask.properties"
    classpath="simian.classpath" classpathref="simian.classpath" />
  <delete dir="${reports.simian.dir}" quiet="true" />
  <mkdir dir="${reports.simian.dir}" />
  <simian threshold="${simian.threshold}" language="${language.type}"
    ignoreCurlyBraces="true" ignoreIdentifierCase="true" ignoreStrings="true"
    ignoreStringCase="true" ignoreNumbers="true"  ignoreCharacters="true">
    <fileset dir="${src.dir}"/>
    <formatter type="xml" toFile="${reports.simian.dir}/${simian.log.file}" />
  </simian>
  <xslt taskname="simian"
    in="${reports.simian.dir}/${simian.log.file}"
    out="${reports.simian.dir}/${simian.report.file}"
    style="${config.dir}/${simian.xsl.file}" />
</target>
```

硬编码的值不仅没有提高灵活性，反而拟制了灵活性。就像在源代码中很容易硬编码数据库连接 `String` 一样，在构建脚本中也应该避免将路径之类的东西硬编码。

测试失败时，构建却能成功

构建远远不止于单纯的源代码编译，它还可能包括自动化开发者测试的执行，如果想让软件一直正常运行，那么决不能允许构建中有任何失败的测试。别忘了，如果测试都得不到信任，那么还要测试干什么呢？

清单 8 是这种构建气味的一个例子。注意 `junit` Ant 任务的 `haltonfailure` 属性被设置为 `false`（它的缺省值）。这意味着即使任何 JUnit 测试是失败的，构建也不会失败。

清单 **8.** 气味：测试失败，构建却成功

```
<junit fork="yes" haltonfailure="false" dir="${basedir}" printsummary="yes">
  <classpath refid="test.class.path" />
  <classpath refid="project.class.path"/>
  <formatter type="plain" usefile="true" />
  <formatter type="xml" usefile="true" />
  <batchtest fork="yes" todir="${logs.junit.dir}">
  <fileset dir="${test.unit.dir}">
    <patternset refid="test.sources.pattern"/>
  </fileset>
  </batchtest>
</junit>
```

有两种方法防止构建中的这种气味。第一种方法是将 `haltonfailure` 属性设置为 `true`。这样就可以防止测试失败构建却成功的情况发生。

对于这种方法，我惟一不喜欢的地方是，我想看看有多大百分比的测试遭到了失败，以便弄清楚失败的模式。因此第二种方法就是，每当有测试失败，就设置一个属性。然后，我对 Ant 进行配置，使得当执行了所有的测试之后，构建最终失败。这两种方法都行之有效。清单 9 演示了使用 `tests.failed` 属性的第二种方法：

清单 **9.** 测试令构建失败

```
<junit dir="${basedir}" haltonfailure="false" printsummary="yes"
  errorProperty="tests.failed" failureproperty="tests.failed">
  <classpath>
    <pathelement location="${classes.dir}" />
  </classpath>
  <batchtest fork="yes" todir="${logs.junit.dir}" unless="testcase">
    <fileset dir="${src.dir}">
      <include name="**/*Test*.java" />
   </fileset>
  </batchtest>
  <formatter type="plain" usefile="true" />
  <formatter type="xml" usefile="true" />
</junit>
<fail if="tests.failed" message="Test(s) failed." />
```

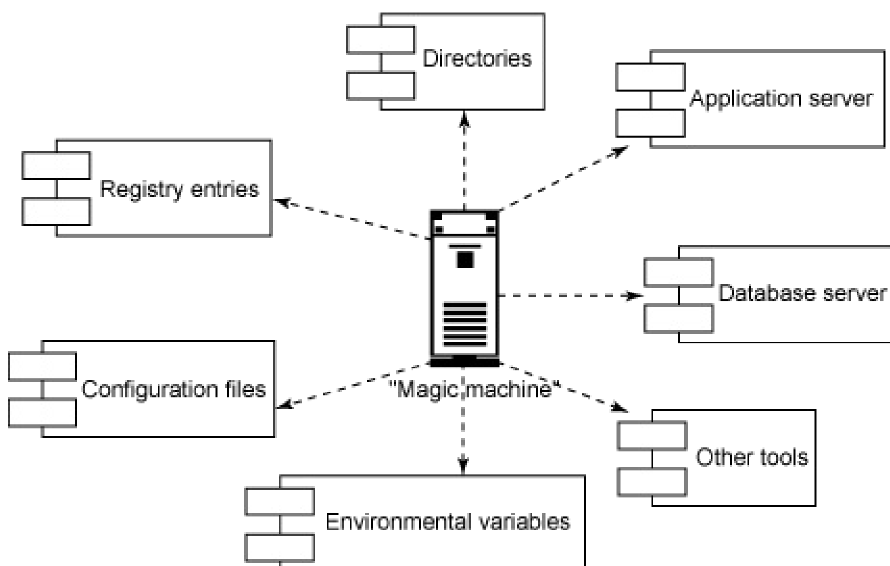
如果测试失败时构建还能通过，就会提供关于安全性的一种错感。如果测试失败，那么让构建也失败：早一点从容地处理问题，总比以后问题半夜三更把您从梦中唤醒要好。

魔力机的气味

在本文谈到的所有气味当中，这一种也许是最难闻的，因为魔力机（*magic machine*）是那种刚好惟一能够构建一个公司的软件应用程序的硬件。这种情况看上去难以相信，实则不然。在我的职业生涯中，就多次碰到过它。当依赖性丢失，或者当不断累积的问题爆发时，这些机器就获得了所谓的魔力。

我们很容易看出，公司基础设施中的一台正常的机器是如何获得魔力的：随着时间的推移，开发者无意间在机器的脚本中添加了硬性的依赖性，包含了对目录路径的全限定引用，甚至安装了只有一台机器上有的工具，久而久之，构建在任何其他机器上再也不能运行了。图 3 就展示了一个例子：

图 **3.** 魔力机



对一台机器的硬编码引用，包括特定驱动器（例如 C：）的路径，以及机器上特有的工具，都是令一台机器着魔的罪魁祸首。每当看到对 C: 盘的引用，或者看到对特定工具（例如 `grep`）的调用时，应该马上更改脚本。如果发现自己声称 "`C:\Program Files\` 目录在每台机器上都有" 的时候，也要三思。

不良格式也有气味

和主流语言中的编程格式一样，在管理构建脚本的时候，也有类似的考虑。当为构建脚本考虑编程格式的时候，需要考虑以下几个方面：

- 属性名称
- 目标名称
- 目录名称
- 环境变量名称
- 缩进
- 代码行长度

就个人而言，对于格式上的约定，我喜欢尽可能利用他人的规则。幸运的是，有人已经提供了那样的参考，即 *The Elements of Ant Style* （见 参考资料）。在这本书中，作者描述了各种规则，例如用小写字母加上用于分隔单词的连字符来命名目标，以及代码行长度和缩进等。不管选择哪一种方法，始终如一地应用有关格式的规则有助于构建文件的长期维护。

构建从来没有如此好闻

我尚能忍受廉价香水的气味。但是，如果说有一样东西我无法忍受的话，那一定是难于维护的构建脚本所散发出的气味。差劲的代码显然会浪费您宝贵的时间，设计不良的构建也不例外。如果构建中还飘散着不一致的、不可重复的和不可维护的气味，那么现在就花时间重构这些至关重要的资源吧。您的开发环境定会香如玫瑰。

追逐代码质量: 决心采用 FIT

试用 *FIT* 和 *JUnit* 进行需求测试工作！

JUnit 假定测试的所有方面都是开发人员的地盘，而集成测试框架（FIT）在编写需求的业务客户和实现需求的开发人员之间做了协作方面的试验。这是否意味着 FIT 和 JUnit 是竞争关系呢？绝对不是！代码质量完美主义者 Andrew Glover 介绍了如何把 FIT 和 JUnit 两者最好的地方结合在一起，实现更好的团队工作和有效的端到端测试。

在软件开发的生命周期中，每个人都对质量负有责任。理想情况下，开发人员在开发周期中，用像 Junit 和 TestNG 这样的测试工具保证早期质量，而质量保证团队用功能性系统测试在周期末端跟进，使用像 Selenium 这样的工具。但是即使拥有优秀的质量保证，有些应用程序在交付的时候仍然被认为是质量低下的。为什么呢？因为它们并没有做它们应当做的事。

在客户、（编写应用程序需求的）业务部门和（实现需求的）开发团队之间的沟通错误，通常是摩擦的原因，有时还是开发项目彻底失败的常见原因。幸运的是，存在一些方法可以帮助需求作者和实现者之间尽早 沟通。

下载 FIT

集成测试框架（FIT）最初是由 Ward Cunningham 创建的，他就是 wiki 的发明人。请访问 Cunningham 的 Web 站点了解关于 FIT 的更多知识并 免费下载它。

FIT 化的解决方案

集成测试框架 （FIT）是一个测试平台，可以帮助需求编写人员和把需求变成可执行代码的人员之间的沟通。使用 FIT，需求被做成表格模型，充当开发人员编写的测试的数据模型。表格本身充当输入和测试的预期输出。

图 1 显示了用 FIT 创建的结构化模型。第一行是测试名称，下一行的三列是与输入（`value1` 和 `value2`）和预期结果（`trend()`）有关的标题。

图 **1.** 用 **FIT** 创建的结构化模型

test.com.acme.fit.impl.TrendIndicator		
value1	*value2*	*trend()*
84.0	71.2	decreasing
67.6	89.0	increasing
50.0	50.0	constant

好消息是，对于编程没有经验的人也能编写这个表格。FIT 的设计目的就是让消费者或业务团队在开发周期中，尽早与实现他们想法的开发人员协作。创建应用程序需求的简单表格式模型，可以让每个人清楚地看出代码和需求是否是一致的。

清单 1 是与图 1 的数据模型对应的 FIT 代码。不要太多地担心细节 —— 只要注意代码有多么简单，而且代码中没有包含验证逻辑（例如，断言等）。可能还会注意到一些与表 1 中的内容匹配的变量和方法名称；关于这方面的内容后面介绍。

清单 **1.** 根据 **FIT** 模型编写的代码

```
package test.com.acme.fit.impl;
import com.acme.sedlp.trend.Trender;
import fit.ColumnFixture;
public class TrendIndicator extends ColumnFixture {
  public double value1;
  public double value2;
  public String trend(){
    return Trender.determineTrend(value1, value2).getName();
  }
}
```

清单 1 中的代码由研究上面表格并插入适当代码的开发人员编写。最后，把所有东西合在一起，FIT 框架读取表 1 的数据，调用对应的代码，并确定结果。

FIT 和 JUnit

FIT 的优美之处在于，它让组织的消费者或业务端能够尽早参与测试过程（例如，在开发期间）。JUnit 的力量在于编码过程中的单元测试，而 FIT 是更高层次的测试工具，用来判断规划的需求实现的正确性。

例如，虽然 JUnit 擅长验证两个 `Money` 对象的合计与它们的两个值的合计相同，但 FIT 可以验证总的订单价格是其中商品的价格减去任何相关折扣之后的合计。区别虽然细微，但的确重大！在 JUnit 示例中，要处理具体的对象（或者需求的实现），但是使用 FIT 时要处理的是

高级的业务过程。

这很有意义，因为编写需求的人通常不太考虑 `Money` 对象 —— 实际上，他们可能根本不知道这类东西的存在！但是，他们确实要考虑，当商品被添加到订单时，总的订单价格应当是商品的价格减去所有折扣。

FIT 和 JUnit 之间绝不是竞争关系，它们是保证代码质量的好搭档，正如在后面的 案例研究 中将要看到的。

测试用的 FIT 表格

表格是 FIT 的核心。有几种不同类型的表格（用于不同的业务场景），FIT 用户可以用不同的格式编写表格。用 HTML 编写表格甚至用 Microsoft Excel 编写都是可以的，如图 2 所示：

图 **2.** 用 **Microsoft Excel** 编写的表格

test.com.acme.fit.impl.TrendIndicator		
value1	*value2*	*trend()*
84.00	71.20	decreasing
67.60	89.00	increasing
50.00	50.00	constant

也有可能用 Microsoft Word 这样的工具编写表格，然后用 HTML 格式保存，如图 3 所示：

图 **3.** 用 **Microsoft Word** 编写的表格

test.com.acme.fit.impl.TrendIndicator		
value1	*value2*	*trend()*
84.0	71.2	decreasing
67.6	89.0	increasing
50.0	50.0	constant

开发人员编写的用来执行表格数据的代码叫作装备（*fixture*）。要创建一个装备类型，必须扩展对应的 FIT 装备，它映射到对应的表。如前所述，不同类型的表映射到不同的业务场景。

用装备进行装配

最简单的表和装备组合，也是 FIT 中最常用的，是一个简单的列表格，其中的列映射到预期过程的输入和输出。对应的装备类型是 `ColumnFixture`。

如果再次查看 清单 1，将注意到 `TrendIndicator` 类扩展了 `ColumnFixture`，而且也与图 3 对应。请注意在图 3 中，第一行的名称匹配完全限定名称（`test.com.acme.fit.impl.TrendIndicator`）。下一行有三列。头两个单元格的值匹配 `TrendIndicator` 类的 `public` 实例成员（`value1` 和 `value2`），最后一个单元格的值只匹配 `TrendIndicator` 中的方法（`trend`）。

现在来看清单 1 中的 `trend` 方法。它返回一个 `String` 值。可以猜测得到，对于表中每个剩下的行，FIT 都会替换值并比较结果。在这个示例中，有三个“数据”行，所以 FIT 运行 `TrendIndicator` 装备三次。第一次，`value1` 被设置成 84.0，`value2` 设置成 71.2。然后 FIT 调用 `trend` 方法，并把从方法得到的值与表中的值比较，应当是 “decreasing”。

通过这种方式，FIT 用装备代码测试 `Trender` 类，每次 FIT 执行 `trend` 方法时，都执行类的 `determineTrend` 方法。当代码测试完成时，FIT 生成如图 4 所示的报告：

图 **4. FIT** 报告 **trend** 测试的结果

test.com.acme.fit.impl.TrendIndicator		
value1	*value2*	*trend()*
84.0	71.2	decreasing
67.6	89.0	increasing
50.0	50.0	constant

trend 列单元格的绿色表明测试通过（例如，FIT 设置 `value1` 为 84.0，`value2` 为 71.2，调用 `trend` 得到返回值 “decreasing”）。

查看 FIT 运行

可以通过命令行，用 Ant 任务并通过 Maven 调用 FIT，从而简单地把 FIT 测试插入构建过程。因为自动进行 FIT 测试，就像 JUnit 测试一样，所以也可以定期运行它们，例如在持续集成系统中。

最简单的命令行运行器，如清单 2 所示，是 FIT 的 `FolderRunner`，它接受两个参数 —— 一个是 FIT 表格的位置，一个是结果写入的位置。不要忘记配置类路径！

清单 **2. FIT** 的命令行

```
%>java fit.runner.FolderRunner ./test/fit ./target/
```

FIT 通过插件，还可以很好地与 Maven 一起工作，如清单 3 所示。只要下载插件，运行 `fit:fit` 命令，就 OK 了！（请参阅 参考资料 获得 Maven 插件。）

清单 **3. Maven** 得到 **FIT**

```
C:\dev\proj\edoa>maven fit:fit
 __  __
|  \/  |__ _Apache__ ___
| |\/| / _` \ V / -_) ' \  ~ intelligent projects ~
|_|  |_\__,_|\_/\___|_||_|  v. 1.0.2
build:start:
java:prepare-filesystem:
java:compile:
    [echo] Compiling to C:\dev\proj\edoa/target/classes
java:jar-resources:
test:prepare-filesystem:
test:test-resources:
test:compile:
fit:fit:
    [java] 2 right, 0 wrong, 0 ignored, 0 exceptions
BUILD SUCCESSFUL
Total time: 4 seconds
Finished at: Thu Feb 02 17:19:30 EST 2006
```

试用 **FIT**：案例研究

现在已经了解了 FIT 的基础知识，我们来做一个练习。如果还没有 下载 FIT，现在是下载它的时候了！如前所述，这个案例研究显示出可以容易地把 FIT 和 JUnit 测试组合在一起，形成多层质量保证。

假设现在要为一个酿酒厂构建一个订单处理系统。酿酒厂销售各种类型的酒类，但是它们可以组织成两大类：季节性的和全年性的。因为酿酒厂以批发方式运作，所以酒类销售都是按桶销售的。对于零售商来说，购买多桶酒的好处就是折扣，而具体的折扣根据购买的桶数和酒是季节性还是全年性的而不同。

麻烦的地方在于管理这些需求。例如，如果零售店购买了 50 桶季节性酒，就没有折扣；但是如果这 50 桶不是 季节性的，那么就有 12% 的折扣。如果零售店购买100 桶季节性酒，那就有折扣，但是只有 5%。100 桶更陈的非季节性酒的折扣达到 17%。购买量达到 200 时，也有类似的规矩。

对于开发人员，像这样的需求集可能让人摸不着头脑。但是请看，我们的啤酒-酿造行业分析师用 FIT 表可以很容易地描述出这个需求，如图 5 所示：

图 **5.** 我的业务需求非常清晰！

org.acme.store.discount.DiscountStructureFIT				
list price per case	*number of cases*	*is seasonal*	*discount price()*	*discount amount()*
$10.00	10	TRUE	$100.00	$0.00
$10.00	10	FALSE	$95.00	$5.00
$10.00	50	TRUE	$500.00	$0.00
$10.00	50	FALSE	$440.00	$60.00
$10.00	100	TRUE	$950.00	$50.00
$10.00	100	FALSE	$830.00	$170.00
$10.00	200	TRUE	$1,800.00	$200.00
$10.00	200	FALSE	$1,600.00	$400.00

表格语义

这个表格从业务的角度来说很有意义，它确实很好地规划出需求。但是作为开发人员，还需要对表格的语言了解更多一些，以便从表格得到值。首先，也是最重要的，表格中的初始行说明表格的名称，它恰好与一个匹配的类对应（`org.acme.store.discount.DiscountStructureFIT`）。命名要求表格作者和开发人员之间的一些协调。至少，需要指定完全限定的表格名称（也就是说，必须包含包名，因为 FIT 要动态地装入对应的类）。

请注意表格的名称以 *FIT* 结束。第一个倾向可能是用 *Test* 结束它，但要是这么做，那么在自动环境中运行 FIT 测试和 JUnit 测试时，会与 JUnit 产生些冲突，JUnit 的类通常通过命名模式查找，所以最好避免用 *Test* 开始或结束 FIT 表格名称。

下一行包含五列。每个单元格中的字符串都特意用斜体格式，这是 FIT 的要求。前面学过，单元格名称与装备的实例成员和方法匹配。为了更简洁，FIT 假设任何值以括号结束的单元格是方法，任何值不以括号结束的单元格是实例成员。

特殊智能

FIT 在处理单元格的值，进行与对应装备类的匹配时，采用智能解析。如 图 5 所示，第二行单元格中的值是用普通的英文编写的，例如 “number of cases”。FIT 试图把这样的字符串按照首字母大写方式连接起来；例如，“number of cases” 变成 “ `numberOfCases` ”，然后 FIT 试图找到对应的装备类。这个原则也适用于方法 —— 如图 5 所示，“discount price()” 变成了 “ `discountPrice()` ”。

FIT 还会智能地猜测单元格中值的具体类型。例如，在 图 5 余下的八行中，每一列都有对应的类型，或者可以由 FIT 准确地猜出，或者要求一些定制编程。在这个示例中，图 5 有三种不同类型。与 “number of cases” 关联的列匹配到 `int` ，而与 “is seasonal” 列关联的值则匹配成 `boolean` 。

剩下的三列，“list price per case”、“discount price()” 和 “discount amount()” 显然代表当前值。这几列要求定制类型，我将把它叫作 `Money` 。有了它之后，应用程序就要求一个代表钱的对象，所以在我的 FIT 装备中遵守少量语义就可以利用上这个对象！

FIT 语义总结

表 1 总结了命名单元格和对应的装备实例变量之间的关系：

表 **1.** 单元格到装备的关系：实例变量

单元格值	对应的装备实例变量	类型
list price per case	`listPricePerCase`	`Money`
number of cases	`numberOfCases`	`int`
is seasonal	`isSeasonal`	`boolean`

表 2 总结了 FIT 命名单元格和对应的装备方法之间的关系：

表 **2.** 单元格到装备的关系：方法

表格单元格的值	对应的装备方法	返回类型
discount price()	`discountPrice`	`Money`
discount amount()	`discountAmount`	`Money`

该构建了！

要为酿酒厂构建的订单处理系统有三个主要对象：一个 `PricingEngine` 处理包含折扣的业务规则，一个 `WholeSaleOrder` 代表订单，一个 `Money` 类型代表钱。

Money 类

第一个要编写的类是 `Money` 类，它有进行加、乘和减的方法。可以用 JUnit 测试新创建的类，如清单 14 所示：

清单 **4. JUnit** 的 **MoneyTest** 类

```
package org.acme.store;
import junit.framework.TestCase;
public class MoneyTest extends TestCase {
  public void testToString() throws Exception{
    Money money = new Money(10.00);
    Money total = money.mpy(10);
    assertEquals("$100.00", total.toString());
  }
  public void testEquals() throws Exception{
    Money money = Money.parse("$10.00");
    Money control = new Money(10.00);
    assertEquals(control, money);
  }
  public void testMultiply() throws Exception{
    Money money = new Money(10.00);
    Money total = money.mpy(10);

    Money discountAmount = total.mpy(0.05);
    assertEquals("$5.00", discountAmount.toString());
  }
  public void testSubtract() throws Exception{
    Money money = new Money(10.00);
    Money total = money.mpy(10);
    Money discountAmount = total.mpy(0.05);
    Money discountedPrice = total.sub(discountAmount);
    assertEquals("$95.00", discountedPrice.toString());
  }
}
```

WholeSaleOrder 类

然后，定义 `WholeSaleOrder` 类型。这个新对象是应用程序的核心：如果 `WholeSaleOrder` 类型配置了桶数、每桶价格和产品类型（季节性或全年性），就可以把它交给 `PricingEngine`，由后者确定对应的折扣并相应地在 `WholeSaleOrder` 实例中配置它。

`WholesaleOrder` 类的定义如清单 5 所示：

清单 **5. WholesaleOrder** 类

```
package org.acme.store.discount.engine;
import org.acme.store.Money;
public class WholesaleOrder {
  private int numberOfCases;
  private ProductType productType;
  private Money pricePerCase;
  private double discount;
  public double getDiscount() {
    return discount;
  }
  public void setDiscount(double discount) {
    this.discount = discount;
  }
  public Money getCalculatedPrice() {
    Money totalPrice = this.pricePerCase.mpy(this.numberOfCases);
    Money tmpPrice = totalPrice.mpy(this.discount);
     return totalPrice.sub(tmpPrice);
  }
  public Money getDiscountedDifference() {
    Money totalPrice = this.pricePerCase.mpy(this.numberOfCases);
    return totalPrice.sub(this.getCalculatedPrice());
  }
  public int getNumberOfCases() {
    return numberOfCases;
  }
  public void setNumberOfCases(int numberOfCases) {
    this.numberOfCases = numberOfCases;
  }
  public void setProductType(ProductType productType) {
    this.productType = productType;
  }
  public String getProductType() {
    return productType.getName();
  }
  public void setPricePerCase(Money pricePerCase) {
    this.pricePerCase = pricePerCase;
  }
  public Money getPricePerCase() {
    return pricePerCase;
  }
}
```

从清单 5 中可以看到，一旦在 `WholeSaleOrder` 实例中设置了折扣，就可以通过分别调用 `getCalculatedPrice` 和 `getDiscountedDifference` 方法得到折扣价格和节省的钱。

更好地测试这些方法（用 **JUnit**）！

定义了 `Money` 和 `WholesaleOrder` 类之后，还要编写 JUnit 测试来验证 `getCalculatedPrice` 和 `getDiscountedDifference` 方法的功能。测试如清单 6 所示：

清单 **6. JUnit** 的 **WholesaleOrderTest** 类

```
package org.acme.store.discount.engine.junit;
import junit.framework.TestCase;
import org.acme.store.Money;
import org.acme.store.discount.engine.WholesaleOrder;
public class WholesaleOrderTest extends TestCase {
  /*
   * Test method for 'WholesaleOrder.getCalculatedPrice()'
   */
  public void testGetCalculatedPrice() {
    WholesaleOrder order = new WholesaleOrder();
    order.setDiscount(0.05);
    order.setNumberOfCases(10);
    order.setPricePerCase(new Money(10.00));
    assertEquals("$95.00", order.getCalculatedPrice().toString());
  }
  /*
   * Test method for 'WholesaleOrder.getDiscountedDifference()'
   */
  public void testGetDiscountedDifference() {
    WholesaleOrder order = new WholesaleOrder();
    order.setDiscount(0.05);
    order.setNumberOfCases(10);
    order.setPricePerCase(new Money(10.00));
    assertEquals("$5.00", order.getDiscountedDifference().toString());
  }
}
```

PricingEngine 类

`PricingEngine` 类利用业务规则引擎，在这个示例中，是 Drools（请参阅“关于 Drools”）。`PricingEngine` 极为简单，只有一个 `public` 方法：`applyDiscount`。只要传递进一个 `WholeSaleOrder` 实例，引擎就会要求 Drools 应用折扣，如清单 7 所示：

清单 7. PricingEngine 类

```
package org.acme.store.discount.engine;
import org.drools.RuleBase;
import org.drools.WorkingMemory;
import org.drools.io.RuleBaseLoader;
public class PricingEngine {
  private static final String RULES="BusinessRules.drl";
  private static RuleBase businessRules;
  private static void loadRules() throws Exception{
    if (businessRules==null){
      businessRules = RuleBaseLoader.
         loadFromUrl(PricingEngine.class.getResource(RULES));
    }
  }
  public static void applyDiscount(WholesaleOrder order) throws Exception{
    loadRules();
    WorkingMemory workingMemory = businessRules.newWorkingMemory( );
    workingMemory.assertObject(order);
    workingMemory.fireAllRules();
  }
}
```

关于 Drools

Drools 是一个为 Java™ 语言度身定制的规则引擎实现。它提供可插入的语言实现，目前规则可以用 Java、Python 和 Groovy 编写。要获得更多信息，或者下载 Drools，请参阅 Drools 主页。

Drools 的规则

必须在特定于 Drools 的 XML 文件中定义计算折扣的业务规则。例如，清单 8 中的代码段就是一个规则：如果桶数大于 9，小于 50，不是季节性产品，则订单有 5% 的折扣。

清单 **8. BusinessRules.drl** 文件的示例规则

```
<rule-set name="BusinessRulesSample"
          xmlns="http://drools.org/rules"
          xmlns:java="http://drools.org/semantics/java"
          xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
          xs:schemaLocation="http://drools.org/rules rules.xsd
                             http://drools.org/semantics/java java.xsd">
<rule name="1st Tier Discount">
  <parameter identifier="order">
    <class>WholesaleOrder</class>
  </parameter>
  <java:condition>order.getNumberOfCases() > 9 </java:condition>
  <java:condition>order.getNumberOfCases() < 50 </java:condition>
  <java:condition>order.getProductType() == "year-round"</java:condition>
  <java:consequence>
    order.setDiscount(0.05);
  </java:consequence>
</rule>
</rule-set>
```

标记团队测试

有了 `PricingEngine` 并定义了应用程序规则之后，可能渴望验证所有东西都工作正确。现在问题就变成，用 JUnit 还是 FIT？为什么不两者都用呢？通过 JUnit 测试所有组合是可能的，但是要进行许多编码。最好是用 JUnit 测试少数几个值，迅速地验证代码在工作，然后依靠 FIT 的力量运行想要的组合。请看看当我这么尝试时发生了什么，从清单 9 开始：

清单 **9. JUnit** 迅速地验证了代码在工作

```
package org.acme.store.discount.engine.junit;
import junit.framework.TestCase;
import org.acme.store.Money;
import org.acme.store.discount.engine.PricingEngine;
import org.acme.store.discount.engine.ProductType;
import org.acme.store.discount.engine.WholesaleOrder;
public class DiscountEngineTest extends TestCase {
  public void testCalculateDiscount() throws Exception{
    WholesaleOrder order = new WholesaleOrder();
    order.setNumberOfCases(20);
    order.setPricePerCase(new Money(10.00));
    order.setProductType(ProductType.YEAR_ROUND);
    PricingEngine.applyDiscount(order);
    assertEquals(0.05, order.getDiscount(), 0.0);
  }
  public void testCalculateDiscountNone() throws Exception{
    WholesaleOrder order = new WholesaleOrder();
    order.setNumberOfCases(20);
    order.setPricePerCase(new Money(10.00));
    order.setProductType(ProductType.SEASONAL);

    PricingEngine.applyDiscount(order);
    assertEquals(0.0, order.getDiscount(), 0.0);
  }
}
```

还没用 FIT？那就用 FIT！

在 图 5 的 FIT 表格中有八行数据值。可能已经在 清单 7 中编写了前两行的 JUnit 代码，但是真的想编写整个测试吗？编写全部八行的测试或者在客户添加新规则时再添加新的测试，需要巨大的耐心。好消息就是，现在有了更容易的方法。不过，不是忽略测试 —— 而是用 FIT！

FIT 对于测试业务规则或涉及组合值的内容来说非常漂亮。更好的是，其他人可以完成在表格中定义这些组合的工作。但是，在为表格创建 FIT 装备之前，需要给 `Money` 类添加一个特殊方法。因为需要在 FIT 表格中代表当前货币值（例如，像 $100.00 这样的值），需要一种方法让 FIT 能够认识 `Money` 的实例。做这件事需要两步：首先，必须把 `static parse` 方法添加到定制数据类型，如清单 10 所示：

清单 10. 添加 parse 方法到 Money 类

```
public static Money parse(String value){
  return new Money(Double.parseDouble(StringUtils.remove(value, '$')));
}
```

`Money` 类的 `parse` 方法接受一个 `String` 值（例如，FIT 从表格中取出的值）并返回配置正确的 `Money` 实例。在这个示例中，`$` 字符被删除，剩下的 `String` 被转变成 `double`，这与 `Money` 中现有的构造函数匹配。

不要忘记向 `MoneyTest` 类添加一些测试来验证新添加的 `parse` 方法按预期要求工作。两个新测试如清单 11 所示：

清单 11. 测试 Money 类的 parse 方法

```
  public void testParse() throws Exception{
    Money money = Money.parse("$10.00");
    assertEquals("$10.00", money.toString());
  }
  public void testEquals() throws Exception{
    Money money = Money.parse("$10.00");
    Money control = new Money(10.00);
    assertEquals(control, money);
 }
```

编写 FIT 装备

现在可以编写第一个 FIT 装备了。实例成员和方法已经在表 1 和表 2 中列出，所以只需要把事情串在一起，添加一两个方法来处理定制类型：`Money`。为了在装备中处理特定类型，还需要添加另一个 `parse` 方法。这个方法的签名与前一个略有不同：这个方法是个对 `Fixture` 类进行覆盖的实例方法，这个类是 `ColumnFixture` 的双亲。

请注意在清单 12 中，`DiscountStructureFIT` 的 `parse` 方法如何比较 `class` 类型。如果存在匹配，就调用 `Money` 的定制 `parse` 方法；否则，就调用父类（`Fixture`）的 `parse` 版本。

清单 12 中剩下的代码是很简单的。对于图 5 所示的 FIT 表格中的每个数据行，都设置值并调用方法，然后 FIT 验证结果！例如，在 FIT 测试的第一次运行中，`DiscountStructureFIT` 的 `listPricePerCase` 被设为 $10.00，`numberOfCases` 设为 10，`isSeasonal` 为 true。然后执行 `DiscountStructureFIT` 的 `discountPrice`，返回的值与 $100.00 比较，然后执行 `discountAmount`，返回的值与 $0.00 比较。

清单 **12.** 用 **FIT** 进行的折扣测试

```
package org.acme.store.discount;
import org.acme.store.Money;
import org.acme.store.discount.engine.PricingEngine;
import org.acme.store.discount.engine.ProductType;
import org.acme.store.discount.engine.WholesaleOrder;
import fit.ColumnFixture;
public class DiscountStructureFIT extends ColumnFixture {
  public Money listPricePerCase;
  public int numberOfCases;
  public boolean isSeasonal;
  public Money discountPrice() throws Exception {
    WholesaleOrder order = this.doOrderCalculation();
    return order.getCalculatedPrice();
  }
  public Money discountAmount() throws Exception {
    WholesaleOrder order = this.doOrderCalculation();
    return order.getDiscountedDifference();
  }
 /**
  * required by FIT for specific types
  */
  public Object parse(String value, Class type) throws Exception {
    if (type == Money.class) {
      return Money.parse(value);
    } else {
      return super.parse(value, type);
    }
  }
  private WholesaleOrder doOrderCalculation() throws Exception {
    WholesaleOrder order = new WholesaleOrder();
    order.setNumberOfCases(numberOfCases);
    order.setPricePerCase(listPricePerCase);
    if (isSeasonal) {
      order.setProductType(ProductType.SEASONAL);
    } else {
      order.setProductType(ProductType.YEAR_ROUND);
    }
    PricingEngine.applyDiscount(order);
    return order;
  }
}
```

现在，比较 清单 9 的 JUnit 测试用例和清单 12。是不是清单 12 更有效率？当然可以 用 JUnit 编写所有必需的测试，但是 FIT 可以让工作容易得多！如果感觉到满意（应当是满意的！），可以运行构建，调用 FIT 运行器生成如图 6 所示的结果：

图 **6.** 这些结果真的很 **FIT** ！

org.acme.store.discount.DiscountStructureFIT				
list price per case	*number of cases*	*is seasonal*	*discount price()*	*discount amount()*
10	10	true	100	0
10	10	false	95	5
10	50	true	500	0
10	50	false	440	60
10	100	true	950	50
10	100	false	830	170
10	200	true	1800	200
10	200	false	1600	400

结束语

FIT 可以帮助企业避免客户和开发人员之间的沟通不畅、误解和误读。把编写需求的人尽早带入测试过程，是在问题成为开发恶梦的根源之前发现并修补它们的明显途径。而且，FIT 与现有的技术（比如 JUnit）完全兼容。实际上，正如本文所示，JUnit 和 FIT 互相补充。请把今年变成您追逐代码质量 的重要纪年 —— 由于决心采用 FIT！

追求代码质量: 不要被覆盖报告所迷惑

您是否曾被测试覆盖度量引入歧途？

测试覆盖工具对单元测试具有重要的意义，但是经常被误用。这个月，Andrew Glover 会在他的新系列 —— 追求代码质量 中向您介绍值得参考的专家意见。第一部分深入地介绍覆盖报告中数字的真实含义。然后他会提出您可以尽早并经常地利用覆盖来确保代码质量的三个方法。

您还记得以前大多数开发人员是如何追求代码质量的吗。在那时，有技巧地放置 `main()` 方法被视为灵活且适当的测试方法。经历了漫长的道路以后，现在自动测试已经成为高质量代码开发的基本保证，对此我很感谢。但是这还不是我所要感谢的全部。Java™ 开发人员现在拥有很多通过代码度量、静态分析等方法来度量代码质量的工具。我们甚至已经设法将重构分类成一系列便利的模式！

要获得有关代码质量问题的答案，您可以访问由 Andrew Glover 主持的 Code Quality 论坛。

所有的这些新的工具使得确保代码质量比以前简单得多，不过您还需要知道如何使用它们。在这个系列中，我将重点阐述有关保证代码质量的一些有时看上去有点神秘的东西。除了带您一起熟悉有关代码质量保证的众多工具和技术之外，我还将为您说明：

- 定义并有效度量最影响质量的代码方面。
- 设定质量保证目标并照此规划您的开发过程。
- 确定哪个代码质量工具和技术可以满足您的需要。
- 实现最佳实践（清除不好的），使确保代码质量及早并经常地 成为开发实践中轻松且有效的方面。

在这个月，我将首先看看 Java 开发人员中最流行也是最容易的质量保证工具包：测试覆盖度量。

谨防上当

这是一个晚上鏖战后的早晨，大家都站在饮水机边上。开发人员和管理人员们了解到一些经过良好测试的类可以达到超过 90% 的覆盖率，正在高兴地互换着 NFL 风格的点心。团队的集体信心空前高涨。从远处可以听到 “放任地重构吧” 的声音，似乎缺陷已成为遥远的记忆，响应性也已微不足道。但是一个很小的反对声在说：

女士们，先生们，不要被覆盖报告所愚弄。

现在，不要误解我的意思：并不是说使用测试覆盖工具是愚蠢的。对单元测试范例，它是很重要的。不过更重要的是您如何理解所得到的信息。许多开发团队会在这儿犯第一个错。

高覆盖率只是表示执行了很多的代码，并不意味着这些代码被很好地 执行。如果您关注的是代码的质量，就必须精确地理解测试覆盖工具能做什么，不能做什么。然后您才能知道如何使用这些工具去获取有用的信息。而不是像许多开发人员那样，只是满足于高覆盖率。

测试覆盖度量

测试覆盖工具通常可以很容易地添加到确定的单元测试过程中，而且结果可靠。下载一个可用的工具，对您的 Ant 和 Maven 构建脚本作一些小的改动，您和您的同事就有了在饮水机边上谈论的一种新报告：测试覆盖报告。当 `foo` 和 `bar` 这样的程序包令人惊奇地显示高 覆盖率时，您可以得到不小的安慰。如果您相信至少您的部分代码可以保证是 “没有 BUG” 的，您会觉得很安心。但是这样做是一个错误。

存在不同类型的覆盖度量，但是绝大多数的工具会关注行覆盖，也叫做语句覆盖。此外，有些工具会报告分支覆盖。通过用一个测试工具执行代码库并捕获整个测试过程中与被 “触及” 的代码对应的数据，就可以获得测试覆盖度量。然后这些数据被合成为覆盖报告。在 Java 世界中，这个测试工具通常是 JUnit 以及名为 Cobertura、Emma 或 Clover 等的覆盖工具。

行覆盖只是指出代码的哪些行被执行。如果一个方法有 10 行代码，其中的 8 行在测试中被执行，那么这个方法的行覆盖率是 80%。这个过程在总体层次上也工作得很好：如果一个类有 100 行代码，其中的 45 行被触及，那么这个类的行覆盖率就是 45%。同样，如果一个代码库包含 10000 个非注释性的代码行，在特定的测试运行中有 3500 行被执行，那么这段代码的行覆盖率就是 35%。

报告分支覆盖 的工具试图度量决策点（比如包含逻辑 `AND` 或 `OR` 的条件块）的覆盖率。与行覆盖一样，如果在特定方法中有两个分支，并且两个分支在测试中都被覆盖，那么您可以说这个方法有 100% 的分支覆盖率。

问题是，这些度量有什么用？很明显，很容易获得所有这些信息，不过您需要知道如何使用它们。一些例子可以阐明我的观点。

代码覆盖在活动

我在清单 1 中创建了一个简单的类以具体表述类层次的概念。一个给定的类可以有一连串的父类，例如 `Vector` ，它的父类是 `AbstractList` ， `AbstractList` 的父类又是 `AbstractCollection` ， `AbstractCollection` 的父类又是 `Object` ：

清单 **1.** 表现类层次的类

```
package com.vanward.adana.hierarchy;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
public class Hierarchy {
  private Collection classes;
  private Class baseClass;
  public Hierarchy() {
    super();
    this.classes = new ArrayList();
  }
  public void addClass(final Class clzz){
    this.classes.add(clzz);
  }
  /**
   * @return an array of class names as Strings
   */
  public String[] getHierarchyClassNames(){
    final String[] names = new String[this.classes.size()];
    int x = 0;
    for(Iterator iter = this.classes.iterator(); iter.hasNext();){
       Class clzz = (Class)iter.next();
       names[x++] = clzz.getName();
    }
    return names;
  }
  public Class getBaseClass() {
    return baseClass;
  }
  public void setBaseClass(final Class baseClass) {
    this.baseClass = baseClass;
  }
}
```

正如您看到的，清单 1 中的 `Hierarchy` 类具有一个 `baseClass` 实例以及它的父类的集合。清单 2 中的 `HierarchyBuilder` 通过两个复制 `buildHierarchy` 的重载的 `static` 方法创建了 `Hierarchy` 类。

清单 2. 类层次生成器

```
package com.vanward.adana.hierarchy;
public class HierarchyBuilder {
  private HierarchyBuilder() {
    super();
  }
  public static Hierarchy buildHierarchy(final String clzzName)
    throws ClassNotFoundException{
      final Class clzz = Class.forName(clzzName, false,
          HierarchyBuilder.class.getClassLoader());
      return buildHierarchy(clzz);
  }
  public static Hierarchy buildHierarchy(Class clzz){
    if(clzz == null){
      throw new RuntimeException("Class parameter can not be null");
    }
    final Hierarchy hier = new Hierarchy();
    hier.setBaseClass(clzz);
    final Class superclass = clzz.getSuperclass();
    if(superclass !=
      null && superclass.getName().equals("java.lang.Object")){
       return hier;
    }else{
       while((clzz.getSuperclass() != null) &&
          (!clzz.getSuperclass().getName().equals("java.lang.Object"))){
             clzz = clzz.getSuperclass();
             hier.addClass(clzz);
       }
       return hier;
    }
  }
}
```

现在是测试时间！

有关测试覆盖的文章怎么能缺少测试案例呢？在清单 3 中，我定义了一个简单的有三个测试案例的 JUnit 测试类，它将试图执行 `Hierarchy` 类和 `HierarchyBuilder` 类：

清单 **3.** 测试 **HierarchyBuilder**！

```
package test.com.vanward.adana.hierarchy;
import com.vanward.adana.hierarchy.Hierarchy;
import com.vanward.adana.hierarchy.HierarchyBuilder;
import junit.framework.TestCase;
public class HierarchyBuilderTest extends TestCase {

  public void testBuildHierarchyValueNotNull() {
     Hierarchy hier = HierarchyBuilder.buildHierarchy(HierarchyBuilderTest.class);
     assertNotNull("object was null", hier);
  }
  public void testBuildHierarchyName() {
     Hierarchy hier = HierarchyBuilder.buildHierarchy(HierarchyBuilderTest.class);
     assertEquals("should be junit.framework.Assert",
       "junit.framework.Assert",
         hier.getHierarchyClassNames()[1]);
  }
  public void testBuildHierarchyNameAgain() {
     Hierarchy hier = HierarchyBuilder.buildHierarchy(HierarchyBuilderTest.class);
     assertEquals("should be junit.framework.TestCase",
       "junit.framework.TestCase",
         hier.getHierarchyClassNames()[0]);
  }

}
```

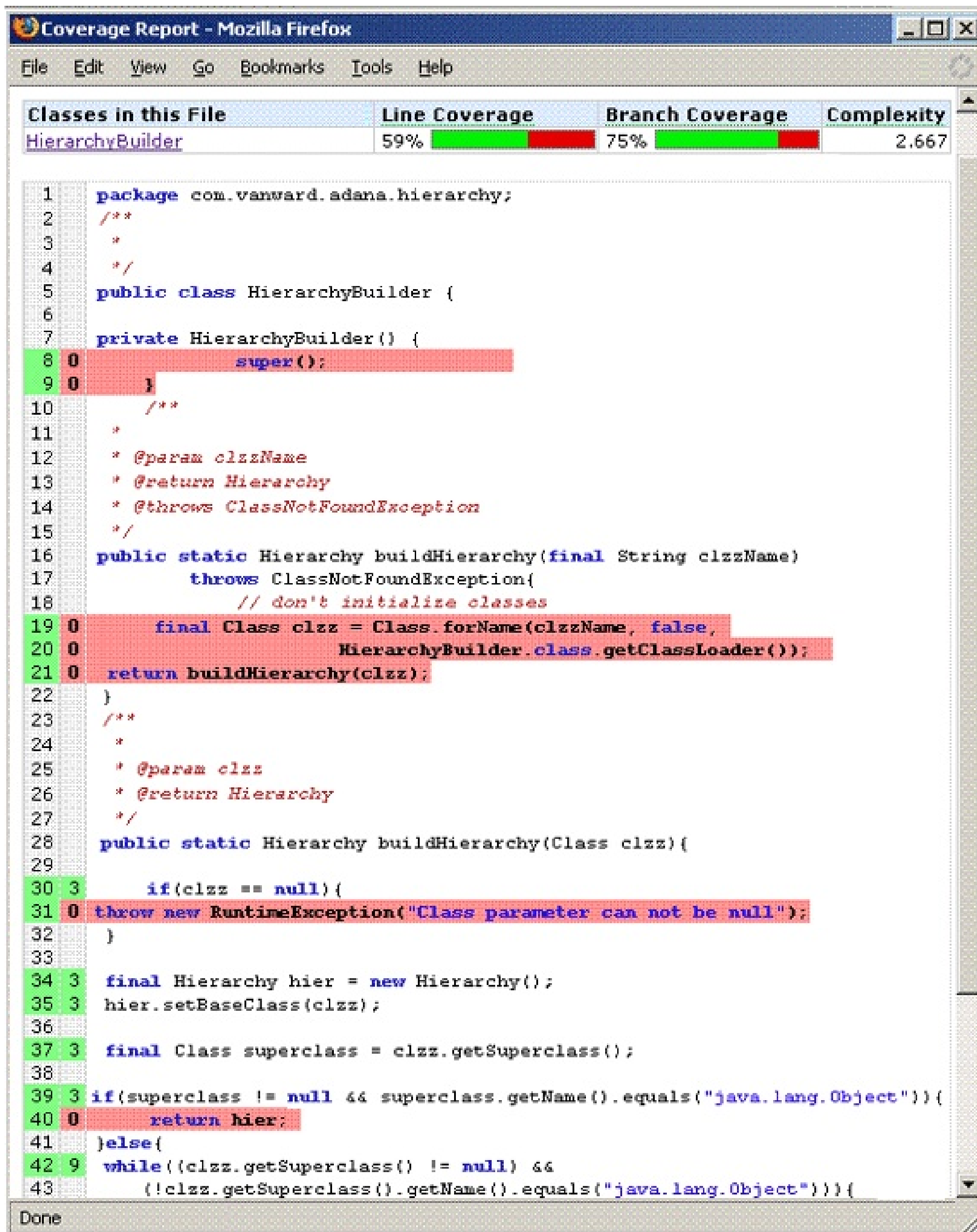
因为我是一个狂热的测试人员，我自然希望运行一些覆盖测试。对于 Java 开发人员可用的代码覆盖工具中，我比较喜欢用 Cobertura，因为它的报告很友好。而且，Corbertura 是开放源码项目，它派生出了 JCoverage 项目的前身。

Cobertura 的报告

运行 Cobertura 这样的工具和运行您的 JUnit 测试一样简单，只是有一个用专门逻辑在测试时检查代码以报告覆盖率的中间步骤（这都是通过工具的 Ant 任务或 Maven 的目标完成的）。

正如您在图 1 中看到的，`HierarchyBuilder` 的覆盖报告说明部分代码没有 被执行。事实上，Cobertura 认为 `HierarchyBuilder` 的行覆盖率为 59%，分支覆盖率为 75%。

图 **1. Cobertura** 的报告



这样看来，我的第一次覆盖测试是失败的。首先，带有 `String` 参数的 `buildHierarchy()` 方法根本没有被测试。其次，另一个 `buildHierarchy()` 方法中的两个条件都没有被执行。有趣的是，所要关注的正是第二个没有被执行的 `if` 块。

因为我所需要做的只是增加一些测试案例，所以我并不担心这一点。一旦我到达了所关注的区域，我就可以很好地完成工作。注意我这儿的逻辑：我使用测试报告来了解什么没有 被测试。现在我已经可以选择使用这些数据来增强测试或者继续工作。在本例中，我准备增强我的测试，因为我还有一些重要的区域未覆盖。

Cobertura：第二轮

清单 4 是一个更新过的 JUnit 测试案例，增加了一些附加测试案例，以试图完全执行 `HierarchyBuilder`：

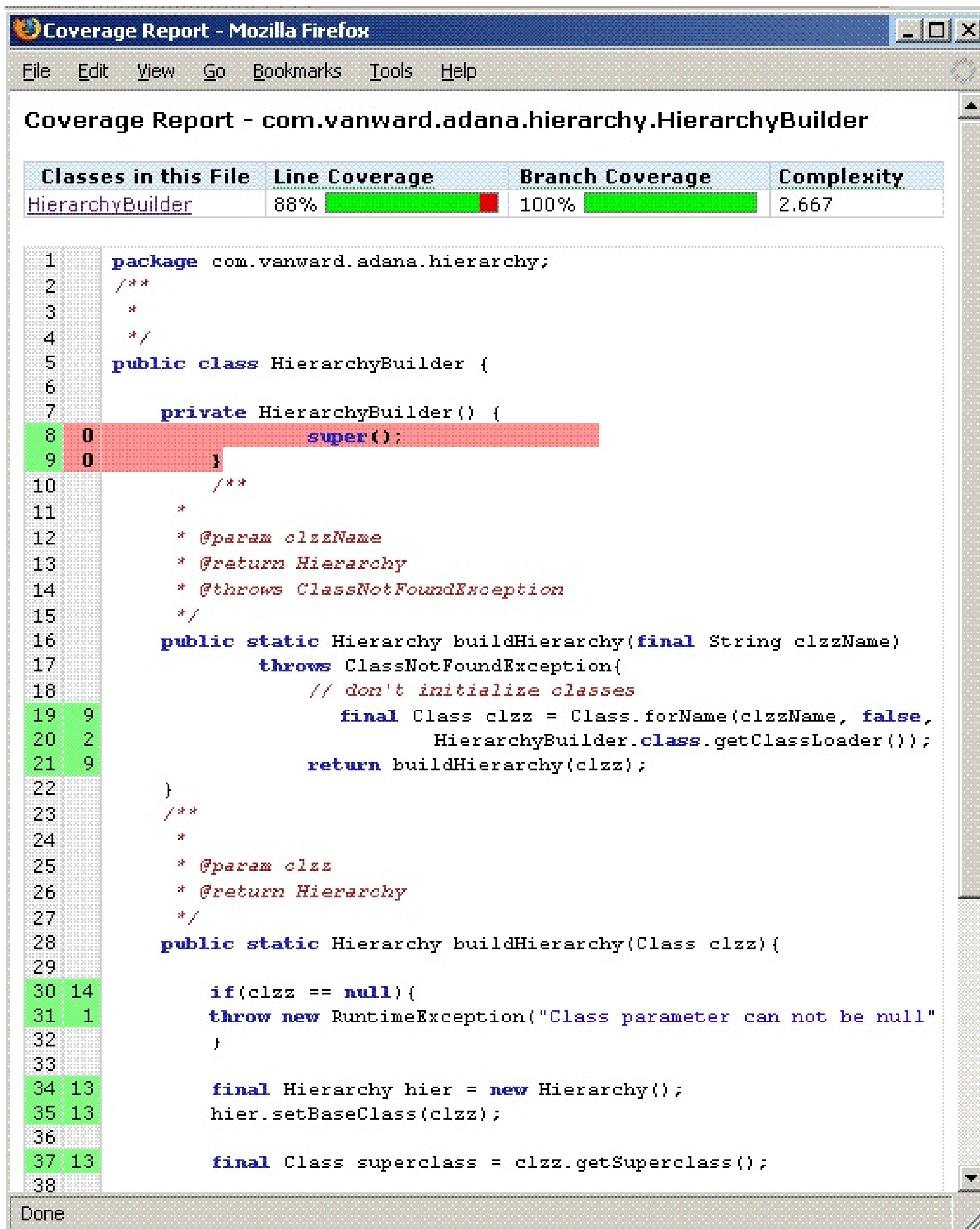
清单 4. 更新过的 JUnit 测试案例

```
package test.com.vanward.adana.hierarchy;
import com.vanward.adana.hierarchy.Hierarchy;
import com.vanward.adana.hierarchy.HierarchyBuilder;
import junit.framework.TestCase;
public class HierarchyBuilderTest extends TestCase {

  public void testBuildHierarchyValueNotNull() {
     Hierarchy hier = HierarchyBuilder.buildHierarchy(HierarchyBuilderTest.class);
     assertNotNull("object was null", hier);
  }
  public void testBuildHierarchyName() {
     Hierarchy hier = HierarchyBuilder.buildHierarchy(HierarchyBuilderTest.class);
     assertEquals("should be junit.framework.Assert",
       "junit.framework.Assert",
         hier.getHierarchyClassNames()[1]);
  }
  public void testBuildHierarchyNameAgain() { zo
     Hierarchy hier = HierarchyBuilder.buildHierarchy(HierarchyBuilderTest.class);
     assertEquals("should be junit.framework.TestCase",
       "junit.framework.TestCase",
         hier.getHierarchyClassNames()[0]);
  }
  public void testBuildHierarchySize() {
     Hierarchy hier = HierarchyBuilder.buildHierarchy(HierarchyBuilderTest.class);
     assertEquals("should be 2", 2, hier.getHierarchyClassNames().length);
  }
  public void testBuildHierarchyStrNotNull() throws Exception{
    Hierarchy hier =
       HierarchyBuilder.
       buildHierarchy("test.com.vanward.adana.hierarchy.HierarchyBuilderTest");
    assertNotNull("object was null", hier);
  }
  public void testBuildHierarchyStrName() throws Exception{
    Hierarchy hier =
       HierarchyBuilder.
       buildHierarchy("test.com.vanward.adana.hierarchy.HierarchyBuilderTest");
    assertEquals("should be junit.framework.Assert",
      "junit.framework.Assert",
        hier.getHierarchyClassNames()[1]);
  }
  public void testBuildHierarchyStrNameAgain() throws Exception{
    Hierarchy hier =
       HierarchyBuilder.
       buildHierarchy("test.com.vanward.adana.hierarchy.HierarchyBuilderTest");
    assertEquals("should be junit.framework.TestCase",
      "junit.framework.TestCase",
        hier.getHierarchyClassNames()[0]);
  }
  public void testBuildHierarchyStrSize() throws Exception{
     Hierarchy hier =
        HierarchyBuilder.
        buildHierarchy("test.com.vanward.adana.hierarchy.HierarchyBuilderTest");
     assertEquals("should be 2", 2, hier.getHierarchyClassNames().length);
  }
  public void testBuildHierarchyWithNull() {
     try{
       Class clzz = null;
       HierarchyBuilder.buildHierarchy(clzz);
       fail("RuntimeException not thrown");
     }catch(RuntimeException e){}
  }
}
```

当我使用新的测试案例再次执行测试覆盖过程时，我得到了如图 2 所示的更加完整的报告。现在，我覆盖了未测试的 `buildHierarchy()` 方法，也处理了另一个 `buildHierarchy()` 方法中的两个 `if` 块。然而，因为 `HierarchyBuilder` 的构造器是 `private` 类型的，所以我不能通过我的测试类测试它（我也不关心）。因此，我的行覆盖率仍然只有 88%。

图 2. 谁说没有第二次机会



正如您看到的，使用一个代码覆盖工具可以 揭露重要的没有相应测试案例的代码。重要的事情是，在阅读报告（特别 是覆盖率高的）时需要小心，它们也许隐含危险的信息。让我们看看两个例子，看看在高覆盖率后面隐藏着什么。

条件带来的麻烦

正如您已经知道的，代码中的许多变量可能有多种状态；此外，条件的存在使得执行有多条路径。在留意这些问题之后，我将在清单 5 中定义一个极其简单只有一个方法的类：

清单 **5.**您能看出下面的缺陷吗？

```
package com.vanward.coverage.example01;
public class PathCoverage {
  public String pathExample(boolean condition){
    String value = null;
    if(condition){
      value = " " + condition + " ";
    }
    return value.trim();
  }
}
```

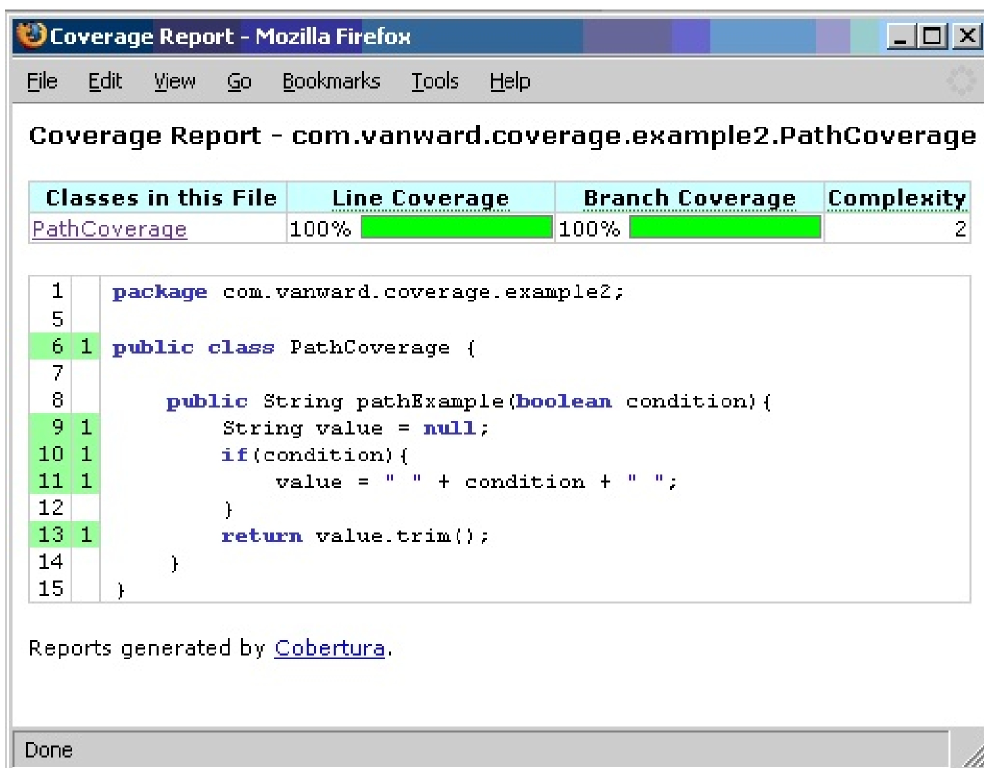
您是否发现了清单 5 中有一个隐藏的缺陷呢？如果没有，不要担心，我会在清单 6 中写一个测试案例来执行 `pathExample()` 方法并确保它正确地工作：

清单 **6. JUnit** 来救援！

```
package test.com.vanward.coverage.example01;
import junit.framework.TestCase;
import com.vanward.coverage.example01.PathCoverage;
public class PathCoverageTest extends TestCase {
  public final void testPathExample() {
    PathCoverage clzzUnderTst = new PathCoverage();
    String value = clzzUnderTst.pathExample(true);
    assertEquals("should be true", "true", value);
  }
}
```

我的测试案例正确运行，我的神奇的代码覆盖报告（如下面图 3 所示）使我看上去像个超级明星，测试覆盖率达到了 100%！

图 **3.** 覆盖率明星



我想现在应该到饮水机边上去说了，但是等等，我不是怀疑代码中有什么缺陷呢？认真检查清单 5 会发现，如果 `condition` 为 `false`，那么第 13 行确实会抛出 `NullPointerException`。*Yeesh*，这儿发生了什么？

这表明行覆盖的确不能很好地指示测试的有效性。

路径的恐怖

在清单 7 中，我定义了另一个包含 *indirect* 的简单例子，它仍然有不能容忍的缺陷。请注意 `branchIt()` 方法中 `if` 条件的后半部分。（`HiddenObject` 类将在清单 8 中定义。）

清单 **7.** 这个代码足够简单

```
package com.vanward.coverage.example02;
import com.acme.someotherpackage.HiddenObject;
public class AnotherBranchCoverage {

  public void branchIt(int value){
    if((value > 100) || (HiddenObject.doWork() == 0)){
      this.dontDoIt();
    }else{
      this.doIt();
    }
  }
  private void dontDoIt(){
    //don't do something...
  }
  private void doIt(){
    //do something!
  }
}
```

呀！清单 8 中的 `HiddenObject` 是有害的。与清单 7 中一样，调用 `doWork()` 方法会导致 `RuntimeException`：

清单 **8.** 上半部分！

```
package com.acme.someotherpackage.HiddenObject;
public class HiddenObject {
  public static int doWork(){
    //return 1;
    throw new RuntimeException("surprise!");
  }
}
```

但是我的确可以通过一个良好的测试捕获这个异常！在清单 9 中，我编写了另一个好的测试，以图挽回我的超级明星光环：

清单 **9.** 使用 **JUnit** 规避风险

```
package test.com.vanward.coverage.example02;
import junit.framework.TestCase;
import com.vanward.coverage.example02.AnotherBranchCoverage;
public class AnotherBranchCoverageTest extends TestCase {

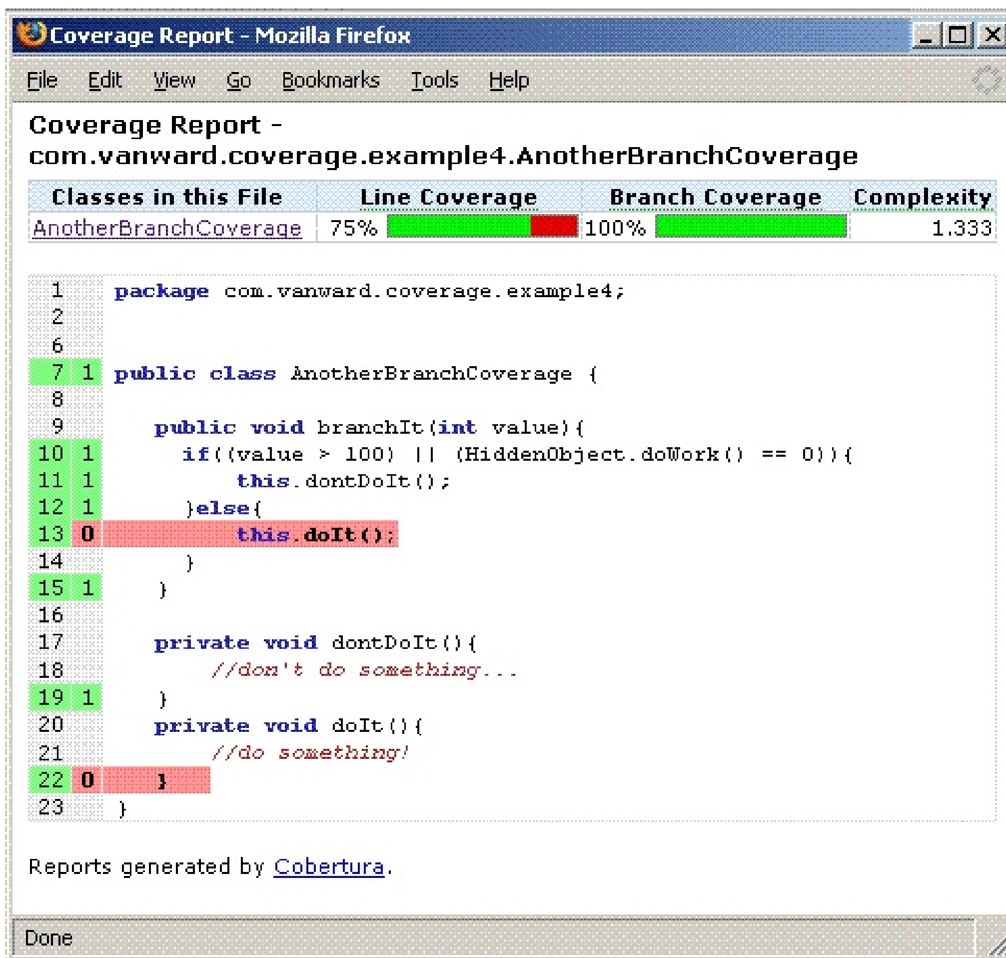
  public final void testBranchIt() {
    AnotherBranchCoverage clzzUnderTst = new AnotherBranchCoverage();
    clzzUnderTst.branchIt(101);
  }
}
```

您对这个测试案例有什么想法？您也许会写出更多的测试案例，但是请设想一下清单 7 中不确定的条件有不止一个的缩短操作会如何。设想如果前半部分中的逻辑比简单的 `int` 比较更复杂，那么您 需要写多少测试案例才能满意？

仅仅给我数字

现在，对清单 7、8、9 的测试覆盖率的分析结果不再会使您感到惊讶。在图 4 的报告中显示我达到了 75% 的行覆盖率和 100% 的分支覆盖率。最重要的是，我执行了第 10 行！

图 **4.**愚弄的报酬



从第一印象看，这让我骄傲。但是这个报告有什么误导吗？只是粗略地看一看报告中的数字，会导致您相信代码是经过良好测试的。基于这一点，您也许会认为出现缺陷的风险很低。这个报告并不能帮助您确定 `or` 缩短操作的后半部分是一个定时炸弹！

质量测试

我不止一次地说：您可以（而且应该）使用测试覆盖工具作为您的测试过程的一部分。但是不要被覆盖报告所愚弄。关于覆盖报告您需要了解的主要事情是，覆盖报告最好用来检查哪些代码没有经过 充分的测试。当您检查覆盖报告时，找出较低的值，并了解为什么特定的代码没有经过充分的测试。知道这些以后，开发人员、管理人员以及 QA 专业人员就可以在真正需要的地方使用测试覆盖工具。通常有下列三种情况：

- 估计修改已有代码所需的时间
- 评估代码质量

- 评定功能测试

现在我可以断定对测试覆盖报告的一些使用方法会将您引入歧途，下面这些最佳实践可以使得测试覆盖报告可以真正为您所用。

1. 估计修改已有代码所需的时间

对一个开发团队而言，针对代码编写测试案例自然可以增加集体的信心。与没有相应测试案例的代码相比，经过测试的代码更容易重构、维护和增强。测试案例因为暗示了代码在测试工作中是如何 工作的，所以还可以充当内行的文档。此外，如果被测试的代码发生改变，测试案例通常也会作相应的改变，这与诸如注释和 Javadoc 这样的静态代码文档不同。

在另一方面，没有经过相应测试的代码更难于理解和安全地 修改。因此，知道代码有没有被测试，并看看实际的测试覆盖数值，可以让开发人员和管理人员更准确地预知修改已有代码所需的时间。

再次回到饮水机边上，可以更好地阐明我的观点。

> 市场部的 Linda：“我们想让系统在用户完成一笔交易时做 *x* 工作。这需要多长时间。我们的用户需要尽快实现这一功能。”
>
> 管理人员 Jeff：“让我看看，这个代码是 Joe 在几个月前编写的，需要对业务层和 UI 做一些变动。Mary 也许可以在两天内完成这项工作。”
>
> Linda：“Joe？他是谁？”
>
> Jeff：“哦，Joe，因为他不知道自己在干什么，所以被我解雇了。”

情况似乎有点不妙，不是吗？尽管如此，Jeff 还是将任务分配给了 Mary，Mary 也认为能够在两天内完成工作 —— 确切地说，在看到代码之前她是这么认为的。

> Mary：“Joe 写这些代码时是不是睡着了？这是我所见过的最差的代码。我甚至不能确认这是 Java 代码。除非推倒重来，要不我根本没法修改。”

情况对 “饮水机” 团队不妙，不是吗？但是我们假设，如果在这个不幸的事件的当初，Jeff 和 Mary 就拥有一份测试报告，那么情况会如何呢？当 Linda 要求实现新功能时，Jeff 做的第一件事就是检查以前生成的覆盖报告。注意到需要改动的软件包几乎没有被覆盖，然后他就会与 Mary 商量。

> Jeff：“Joe 编写的这个代码很差，绝大多数没经过测试。您认为要支持 Linda 所说的功能需要多长时间？”
>
> Mary：“这个代码很混乱。我甚至都不想看到它。为什么不让 Mark 来做呢？”
>
> Jeff：“因为 Mark 不编写测试，刚被我解雇了。我需要您测试这个代码并作一些改动。告诉我您需要多长时间。”
>
> Mary：“我至少需要两天编写测试，然后我会重构这个代码，增加新的功能。我想总共需要四天吧。”

正如他们所说的，知识的力量是强大的。开发人员可以在试图修改代码之前 使用覆盖报告来检查代码质量。同样，管理人员可以使用覆盖数据更好地估计开发人员实际所需的时间。

2. 评估代码质量

开发人员的测试可以降低代码中存在缺陷的风险，因此现在很多开发团队在新开发和更改代码的同时需要编写单元测试。然而正如前面所提到的 Mark 一样，并不总是在编码的同时进行单元测试，因而会导致低质量代码的出现。

监控覆盖报告可以帮助开发团队迅速找出不断增长的没有 相应测试的代码。例如，在一周开始时运行覆盖报告，显示项目中一个关键的软件包的覆盖率是 70%。如果几天后，覆盖率下降到了 60%，那么您可以推断：

- 软件包的代码行增加了，但是没有为新代码编写相应的测试（或者是新增加的测试不能有效地覆盖新代码）。
- 删除了测试案例。
- 上述两种情况都发生了。

能够监控事情的发展，无疑是件好事。定期地查阅报告使得设定目标（例如获得覆盖率、维护代码行的测试案例的比例等）并监控事情的发展变得更为容易。如果您发现测试没有如期编写，您可以提前采取一些行动，例如对开发人员进行培训、指导或帮助。与其让用户“在使用中”发现程序缺陷（这些缺陷本应该在几个月前通过简单的测试暴露出来），或者等到管理人员发现没有编写单元测试时再感到惊讶（和愤怒），还不如采取一些预防性的措施。

使用覆盖报告来确保正确的测试是一项伟大的实践。关键是要训练有素地完成这项工作。例如，使每晚生成并查阅覆盖报告成为连续累计 过程的一部分。

3. 评定功能测试

假设覆盖报告在指出没有经过 足够测试的代码部分方面非常有效，那么质量保证人员可以使用这些数据来评定与功能测试有关的关注区域。让我们回到“饮水机”团队来看看 QA 的负责人 Drew 是如何评价 Joe 的代码的：

> Drew 对 Jeff 说：“我们为下一个版本编写了测试案例，我们注意到很多代码没有被覆盖。那好像是与股票交易有关的代码。”
>
> Jeff：“哦，我们在这个领域有好些问题。如果我是一个赌徒的话，我会对这个功能区域给予特别的关注。Mary 正在对这个应用程序做一些其他的修改 —— 她在编写单元测试方面做得很好，但是这个代码也太差了点。”
>
> Drew：“是的，我正在确定工作的资源和级别，看上去我没必要那么担心了，我估计我们的团队会对股票交易模块引起足够的关注。”

知识再次显示了其强大的力量。与其他软件生命周期中的风险承担者（例如 QA）配合，您可以利用覆盖报告所提供的信息来降低风险。在上面的场景中，也许 Jeff 可以为 Drew 的团队提供一个早期的不包含 Mary 的所有修改的版本。不过无论如何，Drew 的团队都应该关注应用程序的股票交易方面，与其他具有相应单元测试的代码相比，这个地方似乎存在更大的缺陷风险。

测试有什么好处

对单元测试范例而言，测试覆盖度量工具是一个有点奇怪的组成部分。对于一个已存在的有益的过程，覆盖度量可以增加其深度和精度。然而，您应该仔细地阅读代码覆盖报告。单独的高覆盖率并不能确保代码的质量。对于减少缺陷，代码的高覆盖并不是必要条件，尽管高覆盖的代码的确更少 有缺陷。

测试覆盖度量的窍门是使用覆盖报告找出未经 测试的代码，分别在微观和宏观两个级别。通过从顶层开始分析您的代码库，以及分析单个类的覆盖，可以促进深入的覆盖测试。一旦您能够综合这些原则，您和您的组织就可以在真正需要的地方使用覆盖度量工具，例如估计一个项目所需的时间，持续监控代码质量以及促进与 QA 的协作。